Bar recursion is not computable via iteration

Bar recursion is not computable via iteration

John Longley

June 25, 2018

Abstract

We show that the *bar recursion* operators of Spector and Kohlenbach, considered as third-order functionals acting on total arguments, are not computable in Gödel's System T plus minimization, which we show to be equivalent to a programming language with a higher-order iteration construct. The main result is formulated so as to imply the non-definability of bar recursion in $\mathrm{T} + \mathrm{min}$ within a variety of partial and total models, for instance the Kleene-Kreisel continuous functionals. The paper thus supplies proofs of some results stated in the book by Longley and Normann.

The proof of the main theorem makes serious use of the theory of *nested sequential procedures* (also known as PCF Böhm trees), and proceeds by showing that bar recursion cannot be represented by any sequential procedure within which the tree of nested function applications is well-founded.

1 Introduction

In the study of computability theory in a higher-order setting, where 'computable operations' may themselves be passed as arguments to other computable operations, considerable interest attaches to questions of the relative power of different programming languages or other formalisms for computation [20]. In this paper, we shall compare the expressive power of a higher-order language supporting general *iteration* (in the sense of *while* loops) with one supporting general *recursion* (as in recursive function definitions).

On the one hand, it will be easy to see that our iteration constructs are definable via recursion, so that the second language subsumes the first. On the other hand, there is an example due to Berger [3] of a second-order functional H, informally of type $(\mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp) \to \mathbb{N}_\perp$, which is definable via recursion but not via iteration (see Section 1.1 below). So in this sense at least, we may already say that iteration is weaker than recursion.

However, it is crucial to Berger's example that we are considering the behaviour of H on arbitrary (hereditarily) *partial* arguments rather than just on total ones: indeed, Berger also showed that if we merely ask which functionals of types $(\mathbb{N}^r \to \mathbb{N}) \to \mathbb{N}_\perp$ are representable, then iteration (even in a weak form) turns out to be just as powerful as recursion. One may therefore wonder whether, more generally, iteration and recursion offer equally powerful means for defining operations on 'hereditarily total' arguments. The question is a natural one to ask in a computer science context, since it has sometimes

been suggested that it is only the behaviour of a program on total arguments that is likely to matter for practical purposes (see Plotkin [24]).

The main contribution of this paper is to answer this question in the negative: at third order, there are 'hereditarily total' functionals definable by very simple kinds recursion, but not by even the most general kind of iteration that we can naturally formulate. Indeed, one example of such a functional is the well-known *bar recursion* operator, first introduced by Spector in the context of interpretations of classical analysis [27]. Since bar recursion and its close relatives themselves offer a number of intriguing programming possibilities that are active topics of current research (e.g. within game theory [7, 9] and proof mining [14, 22, 2]), we consider this to be an especially significant example of the expressivity difference between iteration and recursion.

More specifically, we will show that neither Spector's original bar recursion functional nor the variant due to Kohlenbach [13] is computable in a language with 'higher-order iteration', even if we restrict attention to 'hereditarily total' arguments. As we shall see, there is more than one way to make such a statement precise, but we shall formulate our theorem in a robust form which (we shall argue) establishes the above claim in all reasonable senses of interest.[1]

As our framework for computation with recursion, we shall work with Plotkin's well-known language PCF for partial computable functionals [23], in which recursion is embodied by a *fixed point* operator $Y_\sigma : (\sigma \to \sigma) \to \sigma$ for each type σ. For iteration, we shall introduce a bespoke language W with a higher-order *while* construct, and show that it is equivalent in power to Gödel's System T extended with the familiar minimization (i.e. unbounded search) operator min. In fact, both PCF and T + min have precursors and analogues in the earlier literature on higher-order computability, and the study of the relationship between (broadly) 'recursive' and 'iterative' styles of computation turns out to have quite deep historical roots. We now survey some of this history in order to provide some further context for our present work.

1.1 Historical context

In a landmark paper of 1959, Kleene [12] provided the first full-blown generalization of a concept of 'effective computability' to all finite type levels, working within the full set-theoretic type structure S of hereditarily total functionals over $\mathbb{N}$. This consisted of an inductive definition of computations via nine schemes S1–S9, resulting in the identification of a substructure $\mathsf{S}^{\mathsf{KI}} \subset \mathsf{S}$ consisting of what we now call the *Kleene computable* functionals (Kleene himself called them *general recursive*). Kleene's scheme S9, in particular, postulates in effect the existence of a 'universal' computable functional, and this in turn gives rise to a very general form of recursive function definition (see e.g. [20, Section 5.1.2]). Indeed, although Kleene's S1–S9 definition looks superficially very different from Plotkin's PCF, it turns out that in a certain sense, the two formalisms express exactly the same class of algorithms for higher-order computation (see [20,

[1] The main results of this paper were stated in [20] as Theorem 6.3.28 and Corollary 6.3.33, with a reference to a University of Edinburgh technical report [17] for the proof. The present paper is a considerably reworked and expanded version of this report, incorporating some minor corrections, and more fully developing the connection with familiar iteration constructs (hence the change in the title).

Sections 6.2 and 7.1]).[2]

In the same paper, Kleene also considered another notion of computability in which S9 was replaced by a weaker scheme S10 for minimization (= unbounded search), giving rise to a substructure $\mathsf{S}^{\min} \subset \mathsf{S}^{\mathsf{KI}}$ of *μ-computable* functionals (Kleene's terminology was *μ-recursive*). Whereas we can regard S9 as giving us 'general recursion', it is natural to think of S10 as giving us a particularly simple kind of 'iteration': indeed, from a modern perspective, we may say that S1–S8 + S10 corresponds to a certain typed λ calculus W_0^{str} with *strict ground-type iteration*, or equivalently to a language $\mathrm{T}_0^{str} + \min$ with strict ground-type primitive recursion and minimization.

With the spectacles of hindsight, then, we can see that in [12] the stage was already set for a comparison between 'iterative' and 'recursive' flavours of higher-order computation. Indeed, in [12, Section 8], Kleene showed (in effect) that the System T recursor $rec_{\mathtt{N}\to\mathtt{N}}$ (a third-order functional in S) was Kleene computable but not μ-computable. However, Kleene's proof relied crucially on the possibility of applying $rec_{\mathtt{N}\to\mathtt{N}}$ to 'discontinuous' arguments (in particular the second-order functional $\exists^2$ embodying quantification over $\mathbb{N}$); it thus left open the question of whether every $\Psi \in \mathsf{S}^{\mathsf{KI}}$ could be mimicked by some $\Psi' \in \mathsf{S}^{\min}$ if one restricted attention to 'computable' arguments.

Over the next two decades, much of the focus of research shifted from the full set-theoretic model S to the Kleene-Kreisel type structure Ct of *total continuous functionals*, a realm of functionals of a more 'constructive' character than S which was found to be better suited to many metamathematical applications (see e.g. [15]). Once again, the notions of μ-computability and Kleene computability respectively pick out substructures $\mathsf{Ct}^{\min} \subseteq \mathsf{Ct}^{\mathsf{KI}}$ of Ct, and Kreisel in [16, page 133] explicitly posed the question of whether these coincide. (The question is harder to answer here than for S: Kleene's counterexample Ψ can no longer be used, because the necessary discontinuous functionals such as $\exists^2$ are no longer present in Ct.) This question remained open for some years until being answered by Bergstra [4], who used an ingenious construction based on the classical theory of c.e. degrees to produce an example of a third-order functional in $\mathsf{Ct}^{\mathsf{KI}}$ but not in $\mathsf{Ct}^{\min}$. On the face of it, Bergstra's example seems ad hoc, but one can extract from his argument the fact that—once again—the System T recursor $rec_{\mathtt{N}\to\mathtt{N}}$ is Kleene computable but not μ-computable (see [20, Section 8.5.2]). The fact that Kreisel's question remained open for so long in the face of such an 'obvious' counterexample suggests that non-computability results of this kind were not readily accessible to the proof techniques of the time.

Although Bergstra's argument improves on Kleene's in that it does not rely on the presence of discontinuous inputs, it still relies on the existence of *non-computable* second-order functions within Ct. The argument is therefore not as robust as we might like: for example, it does not establish the non-μ-computability of $rec_{\mathtt{N}\to\mathtt{N}}$ within the type structure HEO of *hereditarily effective operations*. For this, the necessary techniques had to await certain developments in the computer science tradition, which, in contrast to the work surveyed so far, tended to concentrate on type structures of hereditarily *partial* functionals rather than total ones.

As far as we are aware, the first study of the relative power of iteration and recursion

[2] Strictly speaking, to obtain this equivalence at the algorithmic level, we need a mild extension of PCF with an operator *byval* as described in Subsection 2.2 below.

in a partial setting was that of Berger [3], who (in effect) compared the languages $\mathrm{T}_0 + \mathrm{min}$ and PCF in terms of the elements of Scott's well-known model PC of *partial continuous functionals* that they define. Specifically, Berger introduced the partial functional $H \in \mathsf{PC}((\mathtt{N}^2 \to \mathtt{N}) \to \mathtt{N})$ defined informally by

$$H \;=\; \lambda g^{\mathtt{N}^2 \to \mathtt{N}}.\; g(0, g(1, g(2, g(\cdots)))) \;,$$

and showed that H is readily definable in PCF (using the recursor $Y_{\mathtt{N}\to\mathtt{N}}$), but not at all in $\mathrm{T}_0 + \mathrm{min}$.

Although Berger concentrated on definability in PC, his argument also applies when we replace PC by the type structure $\mathsf{SF}^{\mathrm{eff}}$ of PCF-computable functionals, yielding a slightly stronger result (this point is explained in [20, Section 6.3]). Thus, Berger's result is apparently the first to show that recursion is stronger than iteration in a sense that might matter to programmers: the behaviour of the functional H cannot be mimicked using ground-type iteration alone, even if we restrict attention to *computable* arguments (which we may here take to mean 'arguments definable in $\mathrm{T}_0 + \mathrm{min}$'). Note, however, that Berger's example, unlike those of Kleene and Bergstra, does emphatically depend on the presence of the element $\perp$ in the models.

Berger's paper provided one of the main inspirations for the study of sublanguages of PCF in the book of Longley and Normann [20]. There, the focus was on modelling the 'algorithms' implicit in PCF programs as *nested sequential procedures* (NSPs), also known as PCF *Böhm trees*. This is a model that had roots in early work of Sazonov [25], but which came into focus in the course of work on game semantics for PCF [1, 11]. In summary, an NSP is a potentially infinite 'decision tree' recording the various function calls (including nested calls) that a higher-order program might make, along with the dependency of its behaviour on the results of such calls (further detail will be given in Section 2.2). One of the main ideas explored in [20, Chapter 6] was that certain sublanguages of PCF can be correlated with certain classes of NSPs: for instance, any NSP p definable in $\mathrm{T}_0^{str} + min$ is *left-bounded* (that is, there is a finite global bound d on the nesting depth of function applications within p), while any NSP definable in $\mathrm{T} + min$ is *left-well-founded* (that is, the tree of nested applications within p is well-founded).

It turns out that such observations, together with some concrete combinatorial analysis of NSP computations, can lead to interesting new non-definability results. For instance, one of the main new results of [20] (Theorem 6.3.27) is that no left-bounded procedure can have the same behaviour as $rec_{\mathtt{N}\to\mathtt{N}}$ when restricted to 'total' second-order arguments; it follows that no program of $\mathrm{T}_0^{str} + \mathrm{min}$ can faithfully represent the total functional $rec_{\mathtt{N}\to\mathtt{N}}$ on all 'total' computable inputs. As will become clear below, there is some ambiguity here as regards what 'total' ought to mean for NSPs; however, the theorem in [20] was formulated in a robust way so as to be applicable to any reasonable concept of totality. Moreover, it is also straightforward to transfer the result from NSPs to any *total* model with appropriate structure—in this way, we obtain a robust statement of the non-μ-computability of $rec_{\mathtt{N}\to\mathtt{N}}$ in such models [20, Corollary 6.3.33], not relying on the presence of non-computable or non-total arguments, and immediately applicable to a type structure such as HEO.

This seems to offer a satisfactory conclusion to the story as regards the difference between μ-computability and more general (PCF or Kleene) computability. However,

one is at this point tempted to ask whether the gap between these two notions might be closed simply by extending the former with all the System T recursors rec_σ. Thus, a revised version of Kreisel's question might read: Is the system $\mathrm{T} + \mathit{min}$ as powerful as full Kleene computability for the purpose of defining elements of Ct? (In partial settings such as Scott's PC, the corresponding question for $\mathrm{T} + \mathit{min}$ and PCF is already answered negatively by Berger's H functional, since Berger's argument actually suffices for showing that H is not definable in $\mathrm{T} + \mathit{min}$.) Indeed, one may even feel that $\mathrm{T} + \mathit{min}$ is the more natural level at which to pose such questions, given that $\mathrm{T} + \mathit{min}$ corresponds in expressivity to a language W that embodies a very general and natural concept of iteration (as represented by *while* loops, possibly manipulating higher-order data).

Our main purpose in this paper is to show that, in fact, the famous *bar recursion* operator furnishes the desired example of a (third-order) total functional that is Kleene computable (and hence PCF computable) but not $\mathrm{T} + \mathit{min}$ definable. Put briefly, we shall show that bar recursion does for $\mathrm{T} + \mathit{min}$ everything that $rec_{\mathtt{N}\to\mathtt{N}}$ does for $\mathrm{T}_0^{str} + \mathit{min}$ as described above. As a further piece of relevant background, a brief glance at the history of bar recursion is therefore in order.

Whereas the System T recursors rec_σ allow us to construct functions by recursion on the natural numbers, bar recursion offers a powerful principle for defining functions by recursion on well-founded trees (the precise definition will be given in Subsection 2.3). Bar recursion was introduced by Spector [27] as a major plank of his remarkable extension of Gödel's so-called 'Dialectica' interpretation of first-order arithmetic (modulo a double-negation translation) to the whole of classical analysis (i.e. full second-order arithmetic). Spector's motivations were thus proof-theoretic: for instance, System T extended with bar recursion offered a language of total functionals powerful enough to define all functions $\mathbb{N} \to \mathbb{N}$ provably total in classical analysis. Spector's interpretation, and variations on it, continue to this day to be a fruitful source of results in applied proof theory [14].

Since System T itself defines only the provably total functions of first-order arithmetic, it was thus clear at the outset that bar recursion could not be definable within System T. However, these ideas are of little help when we move to languages such as $\mathrm{T} + \mathit{min}$, which defines all Turing computable functions—nor can methods such as diagonalization be used to establish non-definability in the 'partial' setting of $\mathrm{T} + \mathit{min}$. The results of the present paper thus require quite different techniques.

For the purpose of interpreting classical analysis, one requires versions of bar recursion at many different type levels; however, for the purpose of this paper, we may restrict attention to the simplest non-trivial instance of bar recursion (a third-order operation), since this already turns out to be non-computable in $\mathrm{T} + \mathit{min}$. Another subtlety concerns the way in which we represent the well-founded tree over which the recursion takes place. Typically the tree is specified via a functional $F : (\mathtt{N} \to \mathtt{N}) \to \mathtt{N}$ passed as an argument to the bar recursor—however, different ways of representing trees by such functionals have turned out to have different proof-theoretic applications. In this paper we shall consider two possible choices: the one used by Spector, and a variant due to Kohlenbach [13]. As we shall see, the corresponding versions of bar recursion are actually interdefinable relative to $\mathrm{T} + \mathit{min}$, so that the difference is inessential from the point of view of our main result.

Spector's original treatment of bar recursion was syntactic, but it became clear

through work of Scarpellini [26] and Hyland [10] that bar recursors could be viewed as (Kleene computable) functionals within Ct. Thus, bar recursion was very much in the consciousness of workers in Ct in the early 1970s, although it was evidently not obvious at the time that it furnished a rather dramatic example of a Kleene computable but not μ-computable functional. We will show in this paper how the more recent perspective offered by nested sequential procedures helps to make such results accessible.

1.2 Content and structure of the paper

The main purpose of the paper is to show that the bar recursion functional BR, even at the simplest type level of interest and in a somewhat specialized form, is not definable in System $\mathrm{T} + \mathit{min}$. As we shall see, there are various choices involved in making this statement precise, but our formulation will be designed to be robust with respect to such variations. Our argument will be closely patterned on the proof of the analogous result for $\mathrm{T}_0^{str} + \mathit{min}$ and the System T recursor $\mathit{rec}_{\mathtt{N}\to\mathtt{N}}$ (see [20, Theorem 6.3.27]). However, the present proof will also involve some further twists, illustrating some new possibilities for reasoning with nested sequential procedures.

In Section 2 we define the languages mentioned in the above discussion—PCF, $\mathrm{T} + \mathrm{min}$, W and $\mathrm{T}_0^{str} + \mathit{min}$—and establish some basic relationships between them, in particular showing that $\mathrm{T} + \mathrm{min}$ and W are equivalent in expressive power. We then summarize the necessary theory of nested sequential procedures (NSPs), relying heavily on [20] for proofs, and in particular introducing the crucial substructure of *left-well-founded* procedures, which suffices for modelling $\mathrm{T} + \mathrm{min}$ and W. We also explain the concepts of (Spector and Kohlenbach) bar recursion that we shall work with.

Section 3 is devoted to the proof of our main theorem: within the NSP model, no bar recursor can be left-well-founded, hence no program of $\mathrm{T} + \mathrm{min}$ or W can implement bar recursion, even in a weak sense. As mentioned above, the proof will be closely modelled on the corresponding theorem for $\mathit{rec}_{\mathtt{N}\to\mathtt{N}}$: indeed, we shall take the opportunity to explain more fully certain aspects of that proof that were presented rather tersely in [20]. We shall also explain the new ingredients that form part of the present proof.

In Section 4, we show how our theorem for NSPs transfers readily to other models, both partial and total, under relatively mild conditions. As an example, we infer that bar recursion is not $\mathrm{T} + \mathrm{min}$ definable within the type structure Ct of Kleene-Kreisel continuous functionals.

2 Definitions and prerequisites

In this section we summarize the necessary technical background and establish a few preliminary results. We introduce the languages in question in Subsection 2.1, the nested sequential procedure model in Subsection 2.2, and bar recursion in Subsection 2.3.

2.1 Some languages for recursion and iteration

We start by giving operational definitions of the languages we shall study— principally PCF, $\mathrm{T} + \mathrm{min}$ and W—and establishing some basic relationships between them. A rela-

tively easy result here will be that T+min and W are equally expressive as sublanguages of PCF. Of these, T + min is of course the more widely known and has served as the vehicle for previous results in the area (e.g. in [20]); however, W appears to correspond very directly to a familiar concept of iteration via *while* loops, suggesting that this level of expressivity is a natural one to consider from the perspective of programming language theory.

Our version of PCF will closely follow that of [20, Chapter 7], except that we shall also include product types, at least initially. We shall present all our languages as extensions of a common *base language* B.

Specifically, our types σ are generated by

$$\sigma, \tau \ ::= \ \mathtt{N} \ \mid \ \sigma \to \tau \ \mid \ \sigma \times \tau \ ,$$

Terms of B will be those of the simply typed λ-calculus (with binary products) constructed from the constants

$$\begin{array}{rcll} \widehat{n} & : & \mathtt{N} & \text{for each } n \in \mathbb{N} \\ \mathit{suc},\ \mathit{pre} & : & \mathtt{N} \to \mathtt{N} & \\ \mathit{ifzero} & : & \mathtt{N} \to \mathtt{N} \to \mathtt{N} \to \mathtt{N} & \end{array}$$

Throughout the paper, we shall regard the type of a variable x as intrinsic to x, and will often write x^σ to indicate that x carries the type σ.

We endow B with a call-by-name operational semantics via the following (small-step) basic reduction rules:

$$\begin{array}{rclcrcl} \mathit{fst}\,\langle M, N\rangle & \rightsquigarrow & M & \qquad & \mathit{snd}\,\langle M, N\rangle & \rightsquigarrow & N \\ (\lambda x.M)N & \rightsquigarrow & M[x \mapsto N] & & \mathit{suc}\,\widehat{n} & \rightsquigarrow & \widehat{n+1} \\ \mathit{pre}\,\widehat{n+1} & \rightsquigarrow & \widehat{n} & & \mathit{pre}\,\widehat{0} & \rightsquigarrow & \widehat{0} \\ \mathit{ifzero}\,\widehat{0} & \rightsquigarrow & \lambda xy.x & & \mathit{ifzero}\,\widehat{n+1} & \rightsquigarrow & \lambda xy.y \end{array}$$

We furthermore allow these reductions to be applied in certain term contexts. Specifically, the relation $\rightsquigarrow$ is inductively generated by the basic rules above together with the clause: if $M \rightsquigarrow M'$ then $E[M] \rightsquigarrow E[M']$, where $E[-]$ is one of the *basic evaluation contexts*

$$[-]N \qquad \mathit{suc}\,[-] \qquad \mathit{pre}\,[-] \qquad \mathit{ifzero}\,[-] \qquad \mathit{fst}\,[-] \qquad \mathit{snd}\,[-]\ .$$

We shall consider extensions of B with operations embodying various principles of general recursion, iteration, primitive recursion and minimization. To these we also add some simple operations that allow us to pass arguments of type N 'by value', for reasons we will explain shortly. The operations we consider are given as constants

$$\begin{array}{rcl} Y_\sigma & : & (\sigma \to \sigma) \to \sigma \\ \mathit{while}_\sigma & : & (\sigma \to \mathtt{N}) \to \sigma \to (\sigma \to \sigma) \to \sigma \\ \mathit{rec}_\sigma & : & \sigma \to (\sigma \to \mathtt{N} \to \sigma) \to \mathtt{N} \to \sigma \\ \mathit{min} & : & (\mathtt{N} \to \mathtt{N}) \to \mathtt{N} \to \mathtt{N} \\ \mathit{byval}^{\vec{\sigma}}_\tau & : & (\vec{\sigma} \to \mathtt{N} \to \tau) \to (\vec{\sigma} \to \mathtt{N} \to \tau) \end{array}$$

(where $\vec{\sigma} \to \rho$ abbreviates $\sigma_0 \to \cdots \to \sigma_{r-1} \to \rho$ if $\vec{\sigma} = \sigma_0, \ldots, \sigma_{r-1}$), with associated basic reduction rules

$$\begin{array}{rcl} Y_\sigma\ F & \rightsquigarrow & F(Y_\sigma F) \\ \mathit{while}_\sigma\ C\ X\ F & \rightsquigarrow & \mathit{ifzero}\ (CX)\ (\mathit{while}_\sigma\ C\ (FX)\ F)\ X \\ \mathit{rec}_\sigma\ X\ F\ \widehat{0} & \rightsquigarrow & X \\ \mathit{rec}_\sigma\ X\ F\ \widehat{n+1} & \rightsquigarrow & F\,(\mathit{rec}_\sigma\ X\ F\ \widehat{n})\ \widehat{n} \\ \mathit{min}\ F\ \widehat{n} & \rightsquigarrow & \mathit{ifzero}\ (F\,\widehat{n})\ \widehat{n}\ (\mathit{min}\ F\ (\mathit{suc}\ \widehat{n})) \\ \mathit{byval}^{\vec{\sigma}}_\tau\ F\ \vec{X}\ \widehat{n} & \rightsquigarrow & F\ \vec{X}\ \widehat{n}\ , \quad \text{where } |\vec{X}| = |\vec{\sigma}| \end{array}$$

and with our repertoire of basic evaluation contexts augmented by

$$\mathit{rec}_\sigma\ X\ F\ [-] \qquad \mathit{min}\ F\ [-] \qquad \mathit{byval}^{\vec{\sigma}}_\tau\ F\ \vec{X}\ [-] \quad \text{where } |\vec{X}| = |\vec{\sigma}|\ .$$

The constants Y_σ, rec_σ are familiar from PCF and System T respectively, whilst *min* is an inessential variant of the standard minimization operator μ. The operator while_σ is designed to capture that behaviour of a *while* loop that manipulates data of type σ: the argument C is the looping condition (with $\mathtt{N}$ doing duty for the booleans, and 0 as true), X is the initial value of the data, and F is the transformation applied to the data on each iteration. The result returned by $\mathit{while}_\sigma\, C\, F\, X$ is then the final value of the data when the loop terminates (if it does).

The operator *byval* has a different character. The idea is that the evaluation of $\mathit{byval}^{\vec{\sigma}}_\tau\ F\ \vec{X}\ N\ : \tau$ (where $|\vec{X}| = |\vec{\sigma}|$) will proceed by first trying to compute the value $\widehat{n}$ of N, and if this succeeds, will then call $F\ \vec{X}$ 'by value' on $\widehat{n}$. This ensures that the subterm N will be evaluated exactly once, whereas in the evaluation of $F\ \vec{X}\ N$, N may be evaluated zero times (e.g. if $F = \lambda\vec{x}n.\widehat{0}$) or many times (e.g. if $F = \lambda\vec{x}n.\,\mathit{ifzero}\,n\,n\,n$). It turns out that we need to augment PCF with such operators if we are to simulate the intended reduction behaviours of rec_σ and *min* correctly (see the definitions of Rec_σ and *Min* below). Closely related to this, the addition of *byval* to PCF is needed if we wish to define all computable elements in the NSP model (see Subsection 2.2): there are semantically possible 'evaluation behaviours' that cannot be expressed in PCF alone.

Our operators $\mathit{byval}^{\vec{\sigma}}_\tau$ do essentially the same job as the operator *byval* of [20, Section 7.1], which in our present notation would be written as $\mathit{byval}^{\epsilon}_{\mathtt{N}}$. Indeed, for many purposes one could identify $\mathit{byval}^{\vec{\sigma}}_\tau$ with

$$\lambda f\vec{x}n\vec{y}.\ \mathit{byval}^{\epsilon}_{\mathtt{N}}\,(\lambda n'.\,f\vec{x}n'\vec{y})\ n\ ,$$

but there is a fine-grained difference in reduction behaviour which will matter for Proposition 2 below.

Our languages of interest are obtained by extending the definition of B as follows:

- For PCF, we add the constant Y_σ for each type σ.
- For W, we add the constants while_σ.
- For T, we add the constants rec_σ.

We shall also consider the further extensions T + *min* and PCF + *byval*, where we take the latter to include all $\mathit{byval}^{\vec{\sigma}}_{\tau}$.

Note that in each case, the reduction relation is generated inductively from the specified basic reduction rules together with the clause 'if $M \rightsquigarrow M'$ then $E[M] \rightsquigarrow E[M']$', where $E[-]$ ranges over the appropriately augmented set of basic evaluation contexts. We thus obtain reduction relations $\rightsquigarrow_{\mathrm{PCF}}$, $\rightsquigarrow_{\mathrm{W}}$ etc. on the relevant sets of terms. However, we may, without risk of ambiguity, write $\rightsquigarrow$ for the union of all these reduction relations, noting that $\rightsquigarrow$ is still deterministic, and that if M belongs to one of our languages $\mathcal{L}$ and $M \rightsquigarrow M'$ then M' belongs to $\mathcal{L}$.

We write $\rightsquigarrow^+$ for the transitive closure of $\rightsquigarrow$, and $\rightsquigarrow^*$ for its reflexive-transitive closure. It is easy to see that if M is any closed term of type $\mathtt{N}$, then either $M \rightsquigarrow^* \widehat{n}$ for some unique $n \in \mathbb{N}$, or the unique reduction path starting from M is infinite; in the latter case we say that M *diverges*.

The following fundamental fact will be useful. It was first proved by Milner [21] for PCF, but extends readily to PCF + *byval* (cf. [20, Subsection 7.1.4]). Recall that two closed PCF + *byval* terms $M, M' : \sigma$ are *observationally equivalent* (written $M \simeq_{\mathrm{obs}} M'$) if for every program context $C[-] : \mathtt{N}$ of PCF + *byval* (with a hole of type σ) and every $n \in \mathbb{N}$, we have $C[M] \rightsquigarrow^* \widehat{n}$ iff $C[M'] \rightsquigarrow^* \widehat{n}$.

Theorem 1 (Context lemma for PCF + *byval***)** *(i) Suppose M, M' are closed terms of type $\sigma_0 \to \cdots \to \sigma_{r-1} \to \tau$. Then $M \simeq_{\mathrm{obs}} M'$ iff for all closed $N_0 : \sigma_0$, ..., $N_{r-1} : \sigma_{r-1}$ we have*

$$M N_0 \ldots N_{r-1} \;\simeq_{\mathrm{obs}}\; M' N_0 \ldots N_{r-1} \,.$$

(ii) For closed $M, M' : \sigma \times \tau$, we have $M \simeq_{\mathrm{obs}} M'$ iff $\mathit{fst}\, M \simeq_{\mathrm{obs}} \mathit{fst}\, M'$ and $\mathit{snd}\, M \simeq_{\mathrm{obs}} \mathit{snd}\, M'$.

(iii) For closed $M, M' : \mathtt{N}$, we have $M \simeq_{\mathrm{obs}} M'$ iff M, M' either both diverge or both evaluate to the same numeral $\widehat{n}$.

Let us also write $\equiv$ for the congruence on PCF + *byval* terms generated by $\rightsquigarrow$ (i.e. the least equivalence relation containing $\rightsquigarrow$ and respected by all term contexts $C[-]$). Clearly if $M \rightsquigarrow M'$ then $M \simeq_{\mathrm{obs}} M'$; hence also if $M \equiv M'$ then $M \simeq_{\mathrm{obs}} M'$. In combination with the context lemma, this provides a powerful tool for establishing observational equivalences.

For any type σ, we write $\bot_\sigma$ for the 'everywhere undefined' program $Y_\sigma(\lambda x^\sigma.x)$. It is not hard to see that if $M : \sigma$ admits an infinite reduction sequence then $M \simeq_{\mathrm{obs}} \bot_\sigma$.

At this point, we may note that the addition of *byval* does not fundamentally affect the expressive power of PCF, since as a simple application of the context lemma, we have

$$\mathit{byval}^{\vec{\sigma}}_{\tau} \;\simeq_{\mathrm{obs}}\; \lambda f \vec{x} n \vec{y}.\; \mathit{ifzero}\, n\, (f \vec{x} n \vec{y})\, (f \vec{x} n \vec{y}) \,.$$

This in turn implies that every PCF + *byval* term is observationally equivalent to a PCF term, and also that it makes no difference to the relation $\simeq_{\mathrm{obs}}$ whether the observing contexts $C[-]$ are drawn from PCF + *byval* or just PCF. Even so, we shall treat *byval* as a separate language primitive rather than as a macro for the above PCF term, since its evaluation behaviour is significantly different (cf. Subsection 2.2 below).

We now show how both W and T + *min* may be translated into PCF + *byval*. To do this, we simply need to provide PCF + *byval* programs of the appropriate types to represent the constants while_σ, rec_σ, *min*. As a first attempt, one might consider natural implementations of these operations along the following lines:

$$\begin{array}{rcl} \mathit{While}_\sigma & = & \lambda c\, x f.\ Y_{\sigma\to\sigma}(\lambda w.\ \mathit{ifzero}\ (c\, x)\ (w(f\, x))\ x) \\ \mathit{Rec}_\sigma & = & \lambda x f.\ Y_{\mathtt{N}\to\sigma}(\lambda r.\lambda n.\ \mathit{ifzero}\ n\ x\ (f(r(\mathit{pre}\, n))(\mathit{pre}\, n))) \\ \mathit{Min} & = & \lambda f.\ Y_{\mathtt{N}\to\mathtt{N}}(\lambda m.\lambda n.\ \mathit{ifzero}\ (f\, n)\ n\ (m\, (\mathit{suc}\, n))) \end{array}$$

In fact, the above program While_σ will serve our purpose as it stands, but for Rec_σ and *Min* we shall need to resort to more complicated programs that mimic the reduction behaviour of rec_σ, *min* in a more precise way, using *byval* to impose a certain evaluation order. We abbreviate the type of rec_σ as ρ, and the type of *min* as μ; we also write byval^+_σ for $\mathit{byval}^{\sigma,(\sigma\to\mathtt{N}\to\sigma)}_\sigma$.

$$\begin{array}{rcl} \mathit{Rec}_\sigma & = & \mathit{byval}^+_\sigma\ (Y_\rho\, (\lambda r^\rho. \\ & & \quad \lambda x f n.\ \mathit{ifzero}\ n\ x\ (\mathit{byval}^\epsilon_{\mathtt{N}}\ (\lambda n'.\ f\ ((\mathit{byval}^+_\sigma\ r) x f n')\ n')\ (\mathit{pre}\, n)))) \\ \mathit{Min} & = & \mathit{byval}^{\mathtt{N}\to\mathtt{N}}_{\mathtt{N}}\ (Y_\mu\, (\lambda m^\mu.\ \lambda f n.\ \mathit{ifzero}\ (f\, n)\ n\ ((\mathit{byval}^{\mathtt{N}\to\mathtt{N}}_{\mathtt{N}}\ m)\ f\ (\mathit{suc}\, n)))) \end{array}$$

We may then translate a term $M : \sigma$ of W or T + *min* to a term $M^\circ : \sigma$ of PCF + *byval* simply by replacing each occurrence of while_σ, rec_σ, *min* by the corresponding PCF + *byval* program. The following facts are routine to check by induction on the generation of $\rightsquigarrow$:

Proposition 2 *(i) If $M \rightsquigarrow M'$ then $M^\circ \rightsquigarrow^+ M'^\circ$.*

(ii) If M° can be reduced, then so can M: that is, if $M^\circ \rightsquigarrow N$ then there is some M' such that $M \rightsquigarrow M'$.

(iii) Hence $M \rightsquigarrow^* \widehat{n}$ iff $M^\circ \rightsquigarrow^* \widehat{n}$. □

Next, we show that W and T + *min* are also intertranslatable, though in a looser sense. First, in either of these languages, it is an easy exercise to write a program $\neq\colon \mathtt{N} \to \mathtt{N} \to \mathtt{N}$ that implements inequality testing. To assist readability, we shall use $\neq$ as an infix, and also allow ourselves some obvious pattern-matching notation for λ-abstractions on product types. To translate from T + *min* to W, we use the W programs

$$\begin{array}{rcl} \mathit{Rec}'_\sigma & = & \lambda x f n.\ \mathit{snd}\ (\mathit{while}_{\mathtt{N}\times\sigma}\ (\lambda\langle n', x'\rangle.\ n' \neq n) \\ & & \qquad\qquad \langle 0, x\rangle\ \ (\lambda\langle n', x'\rangle.\ \langle \mathit{suc}\, n', f x' n'\rangle)) \\ \mathit{Min}' & = & \lambda f n.\ \mathit{while}_{\mathtt{N}}\ (\lambda n'.\ f n' \neq 0)\ n\ \mathit{suc} \end{array}$$

We may then translate a T + *min* term M to a W term $M^\dagger$ simply by replacing each occurrence of rec_σ *min* by Rec'_σ, Min' respectively. However, it will *not* in general be the case for this translation that if $M \rightsquigarrow M'$ then $M^\dagger \rightsquigarrow^+ M'^\dagger$: the operational behaviour of M and $M^\dagger$ at an intensional level may be quite different. Nevertheless, we can show that the translation is faithful in the sense that M and $M^\dagger$ are *observationally equivalent* when both are transported to PCF:

Proposition 3 *(i)* $(Rec'_\sigma)^\circ \simeq_{\mathrm{obs}} Rec_\sigma$ *and* $(Min')^\circ \simeq_{\mathrm{obs}} Min$ *as* PCF *terms.*
(ii) For any closed term M *of* $\mathrm{T}+min$*, we have* $(M^\dagger)^\circ \simeq_{\mathrm{obs}} M^\circ$*.*
(iii) For any closed $M : \mathtt{N}$ *in* $\mathrm{T}+min$*, we have* $M \rightsquigarrow^* n$ *iff* $M^\dagger \rightsquigarrow^* n$*.*

Proof sketch: (i) For Rec_σ, by Theorem 1 it suffices to show that for any closed PCF terms $X : \sigma$, $F : \sigma \to \mathtt{N} \to \sigma$ and $N : \mathtt{N}$, we have

$$(Rec'_\sigma)^\circ\ X\ F\ N\ \simeq_{\mathrm{obs}}\ Rec_\sigma\ X\ F\ N\ .$$

But this is routinely verified: if N diverges then both sides admit infinite reduction sequences and so are observationally equivalent to $\bot_\sigma$; whilst if $N \rightsquigarrow^* n$ then an easy induction on n shows that

$$(Rec'_\sigma)^\circ\ X\ F\ n\ \equiv\ F\,(\cdots(F\,(F\,X\,\widehat{0})\,\widehat{1})\cdots)\,\widehat{n-1}\ \equiv\ Rec_\sigma\ X\ F\ n\ .$$

A similar approach works for *Min*.

(ii) follows immediately, since $(M^\dagger)^\circ$ may be obtained from M° by replacing certain occurrences of Rec_σ, Min by $(Rec'_\sigma)^\circ$, $(Min')^\circ$.

(iii) is now immediate from (ii) and Proposition 2(iii). □

To translate from W to $\mathrm{T}+min$, we may define

$$\begin{aligned} While'_\sigma \quad = \quad & \lambda cxf.\ Rec_\sigma\ x\ (\lambda x'n.\ fx') \\ & \qquad (min\ (\lambda n.\ Rec_\sigma\ x\ (\lambda x'n.\ fx') \neq 0)\ 0) \end{aligned}$$

and translate a W term M to a $\mathrm{T}+min$ term $M^\ddagger$ by replacing each $while_\sigma$ with $While'$. Once again, the operational behaviour of $M^\ddagger$ is in general quite different from that of M: indeed, this translation is grossly inefficient from a practical point of view, since typically some subcomputations will be repeated several times over. Nonetheless, if we consider programs only up to observational equivalence, the translation is still faithful in the way that we require:

Proposition 4 *(i)* $(While'_\sigma)^\circ \simeq_{\mathrm{obs}} While_\sigma$*.*
(ii) For any closed term M *of* W*, we have* $(M^\ddagger)^\circ \simeq_{\mathrm{obs}} M^\circ$*.*
(iii) For any closed $M : \mathtt{N}$ *in* W*, we have* $M \rightsquigarrow^* n$ *iff* $M^\ddagger \rightsquigarrow^* n$*.*

Proof sketch: Closely analogous to Proposition 3. For (i), we show that by induction on n that if $C(F^n(X)) \rightsquigarrow \widehat{0}$ whereas $C(F^i(X)) \rightsquigarrow \widehat{m_i} \neq \widehat{0}$ for each $i < n$, then

$$(While'_\sigma)^\circ\ C\ F\ X\ \equiv\ F^n(X)\ \equiv\ While_\sigma\ C\ F\ X\ .$$

Furthermore, we show that if there is no n with this property, then $(While'_\sigma)^\circ\ C\ F\ X$ and $While_\sigma\ C\ F\ X$ both admit infinite reduction sequences, so that both are observationally equivalent to $\bot_\sigma$. □

In summary, we have shown that $\mathrm{T}+min$ and W are equally expressive as sublanguages of PCF, in the sense that a closed PCF term M is observationally equivalent to (the image of) a $\mathrm{T}+min$ term iff it is observationally equivalent to a W term. As already noted, the language W appears to embody a natural general principle of iteration, suggesting that this level of expressive power is a natural one to consider.

Although not formally necessary for this paper, it is also worth observing that the above equivalence works level-by-level. For each $k \geq 0$, let us define sublanguages PCF_k, $\mathrm{T}_k + \mathit{min}$, W_k of PCF, $\mathrm{T} + \mathit{min}$, W by admitting (respectively) the constants Y_σ, rec_σ, while_σ only for types σ of level $\leq k$. Then the translations $-^\dagger, -^\ddagger$ clearly restrict to translations between $\mathrm{T}_k + \mathit{min}$ and W_k, so that $\mathrm{T}_k + \mathit{min}$ and W_k are equally expressive as sublanguages of PCF.[3] In fact, for $k \geq 1$, we can even regard them as sublanguages of PCF_k, since it is an easy exercise to replace our 'precise' translation $-^\circ$ for $\mathrm{T}_k + \mathit{min}$ or W_k by a translation $-^\bullet$ up to observational equivalence that requires only PCF_k. This does not work for $k = 0$, however, since the implementation of $\mathit{rec}_{\mathtt{N}}$ requires at least $Y_{\mathtt{N}\to\mathtt{N}}$.

2.1.1 Weaker languages

A few weaker languages will play more minor roles in the paper. We introduce them here, suppressing formal justifications of points mentioned merely for the sake of orientation.

To motivate these languages, we note that even the primitive recursor $\mathit{rec}_{\mathtt{N}}$ of T_0 works in a 'lazy' way: it is possible for the value of $\mathit{rec}_{\mathtt{N}}\, X\, F\, \widehat{n+1}$ to be defined even if that of $\mathit{rec}_{\mathtt{N}}\, X\, F\, \widehat{n}$ is not, for example if $n = 0$, $X = \bot_{\mathtt{N}}$ and $F = \lambda x n.0$. This contrasts with the 'strict' behaviour of Kleene's original version of primitive recursion, in which the value of $\mathit{rec}_{\mathtt{N}}\, X\, F$ at $\widehat{n}$ is obtained by successively computing its values at $\widehat{0}, \widehat{1}, \ldots, \widehat{n-1}$, all of which must be defined. (Of course, the distinction is not very visible in purely total settings such as S or Ct.)

We may capture this stricter behaviour with the help of $\mathit{byval}^\epsilon_{\mathtt{N}} : (\mathtt{N} \to \mathtt{N}) \to \mathtt{N} \to \mathtt{N}$, which we shall here write as $\mathit{byval}_{[\mathtt{N}]}$. From this, we may inductively define an operator

$$\mathit{byval}_{[\sigma]} \;:\; (\sigma \to \sigma) \to \sigma \to \sigma$$

for each level 0 type σ by:

$$\mathit{byval}_{[\sigma\times\tau]} \;=\; \lambda f.\, \lambda x^{\sigma\times\tau}.\, \mathit{byval}_{[\sigma]}\, (\lambda y^{\sigma}.\, \mathit{byval}_{[\tau]}(\lambda z^{\tau}.\, f\langle y, z\rangle)\, (\mathit{snd}\, x))\, (\mathit{fst}\, x)\ .$$

We may now use these operators to define 'strict' versions of rec_σ and while_σ for any type σ of level 0. We do this by introducing constants $\mathit{rec}^{\mathit{str}}_\sigma$ and $\mathit{while}^{\mathit{str}}_\sigma$ of the same types as rec_σ, while_σ, with reduction rules

$$\begin{array}{rcl}
\mathit{rec}^{\mathit{str}}_\sigma\ X\ F\ \widehat{0} & \rightsquigarrow & X \\
\mathit{rec}^{\mathit{str}}_\sigma\ X\ F\ \widehat{n+1} & \rightsquigarrow & \mathit{byval}_{[\sigma]}\ (\lambda m.\, F\, m\, \widehat{n})\ (\mathit{rec}^{\mathit{str}}_\sigma\ X\ F\ \widehat{n}) \\
\mathit{while}^{\mathit{str}}_\sigma\ C\ X\ F & \rightsquigarrow & \mathit{byval}_{[\sigma]}\ (\lambda x.\, \mathit{ifzero}\ (Cx)\ (\mathit{while}_\sigma\ C\ (Fx)\ F)\ x)\ X
\end{array}$$

We also add $\mathit{rec}^{\mathit{str}}_\sigma\, X\, F\, [-]$ as a basic evaluation context. By using these in place of their non-strict counterparts, and also including $\mathit{byval}_{[\mathtt{N}]}$ so that the operators $\mathit{byval}_{[\sigma]}$ are available, we obtain languages $\mathrm{T}^{\mathit{str}}_0$, $\mathrm{T}^{\mathit{str}}_0 + \mathit{min}$ and W^{str}_0.

Our earlier PCF programs Rec_σ, While_σ may be readily adapted to yield faithful translations of these languages into PCF + *byval*. It can also be checked that $\mathrm{T}^{\mathit{str}}_0 +$

[3] This hierarchy turns out to be strict, as will be shown in a forthcoming paper [19]. The strictness of the hierarchy of languages PCF_k is established in [18].

min and W_0^{str} are intertranslatable in the same way as $\mathrm{T}_0 + \mathit{min}$ and W_0, and so are equi-expressive as sublanguages of PCF. Finally, we can regard the strict versions as sublanguages of the lazy ones up to observational equivalence,[4] since for example

$$\mathit{rec}_\sigma^{str} \;\simeq_{\mathrm{obs}}\; \lambda x f n.\ \mathit{byval}_{[\sigma]}\ (\lambda x'.\ \mathit{rec}_\sigma\, x\, (\lambda y m.\ \mathit{byval}(\lambda y'.\, f y' n)\, y)\, n)\ x\ .$$

and it is easy to supply terms of B observationally equivalent to each $\mathit{byval}_{[\sigma]}$.

It is natural to think of W_0^{str} as the language for 'everyday' iterative computations on ground data. It is easy to check that at type level 1, $\mathrm{T}_0^{str} + \mathit{min}$ and W_0^{str} define all Turing computable functions, and indeed that (respectively) $\mathit{rec}_{\mathtt{N}}^{str}$ and $\mathit{while}_{\mathtt{N}\times\mathtt{N}}^{str}$ are sufficient for this purpose.

The weakest language of all that we shall consider is T_0^{str}. This will play an ancillary as a language for a rudimentary class of 'non-controversially total' functionals present in all settings of interest, enabling us to formulate our main theorem in a robust and portable form. In all total type structures of interest, the T_0^{str}-definable functionals will clearly coincide with those given by Kleene's S1–S8; at type level 1 these are just the usual primitive recursive functions. However, a somewhat subtle point is that for typical interpretations in *partial* type structures, T_0^{str} will be slightly stronger than the usual formulations of S1–S8, since the constant *ifzero* will give us the power of *strong definition by cases* which is not achievable via S1–S8 alone. This point will be significant in Section 3, where we shall frequently claim that certain elements of the model SP^0 are T_0^{str}-definable; the reader should bear in mind here that strong definitions by cases are permitted.[5]

2.1.2 Elimination of product types

So far, we have worked with languages with product types in order to manifest the equivalence of T + min and W (and various restrictions thereof) in a perspicuous way. However, since the bar recursors that are the subject of our main theorem have a type not involving products, it will be sufficient from here on to work with types without $\times$, and it will simplify the presentation of nested sequential procedures (in the next subsection) to do so. From the point of view of expressivity, nothing of significance is lost by dispensing with product types, in view of the following proposition. Here we say a type σ is $\times$*-free* if it does not involve products, and a term M is $\times$*-free* if the types of M and all its subterms are $\times$-free. In particular, a $\times$-free term M may involve operators Y_σ, rec_σ, while_σ only for $\times$-free σ. We shall write id_σ for $\lambda x^\sigma.x$, and $g \circ f$ for $\lambda x.g(f\,x)$.

Proposition 5 *Suppose σ is $\times$-free. Then any closed term $M : \sigma$ of* PCF *(resp.* T + *min,* W*) is observationally equivalent to a $\times$-free term of the same language.*

[4] It is shown in [20, Theorem 6.3.23] that rec_0^{str} is strictly weaker than rec_0 (note that $\mathrm{T}_0 + \mathit{min}$ is in essence the language known as $\mathsf{Klex}^{\mathrm{min}}$ in [20]). What is perhaps more surprising is that $\mathrm{T}_0 + \mathit{min}$ defines more *total* functionals at third order than does $\mathrm{T}_0^{str} + \mathit{min}$, despite the fact that rec_0, rec_0^{str} have the same behaviour on all total arguments [19].

[5] In [20], this issue was addressed by introducing a specially defined class $\mathsf{SP}^{0,\mathrm{prim}+}$ of *strongly total* elements of SP^0.

PROOF SKETCH: This will be clear from familiar facts regarding the embeddability of arbitrary types in $\times$-free ones, as covered in detail in [20, Section 4.2]. More specifically, we may specify, for each type σ, a $\times$-free type $\widehat{\sigma}$ such that σ is a T_0^{str}-definable retract of $\widehat{\sigma}$ up to observational equivalence: that is, there are closed B terms $enc_\sigma : \sigma \to \widehat{\sigma}$ and $dec_\sigma : \widehat{\sigma} \to \sigma$ such that $\lambda x^\sigma.\, dec_\sigma(enc_\sigma\, x) \simeq_{\mathrm{obs}} id_\sigma$. Moreover, we may choose these data in such a way that

- $\widehat{\mathtt{N}} = \mathtt{N}$ and $\widehat{\sigma \to \tau} = \widehat{\sigma} \to \widehat{\tau}$, and moreover we have $enc_{\mathtt{N}} = dec_{\mathtt{N}} = id_{\mathtt{N}}$, $enc_{\sigma\to\tau} = \lambda f.\, enc_\tau \circ f \circ dec_\sigma$, and $dec_{\sigma\to\tau} = \lambda g.\, dec_\tau \circ g \circ enc_\sigma$ (these facts imply that for all $\times$-free σ we have $\widehat{\sigma} = \sigma$ and $enc_\sigma \simeq_{\mathrm{obs}} dec_\sigma \simeq_{\mathrm{obs}} id_\sigma$),
- pairing and projections are represented relative to this encoding by $\times$-free programs $Pair : \widehat{\sigma} \to \widehat{\tau} \to \widehat{\sigma \times \tau}$, $Fst : \widehat{\sigma \times \tau} \to \widehat{\sigma}$, $Snd : \widehat{\sigma \times \tau} \to \widehat{\tau}$,
- for any σ we have $Y_\sigma \simeq_{\mathrm{obs}} dec_\sigma(Y_{\widehat{\sigma}})$, and similarly for rec_σ and $while_\sigma$.

Using these facts, it is easy to construct a compositional translation assigning to each term $M : \sigma$ (with free variables $x_i : \sigma_i$) a $\times$-free term $\widehat{M} : \widehat{\sigma}$ (with free variables $\widehat{x_i} : \widehat{\sigma_i}$) such that $M \simeq_{\mathrm{obs}} dec_\sigma(\widehat{M}[\widehat{\vec{x}} \mapsto enc(\vec{x})])$ (we omit the uninteresting details). In particular, for closed M of $\times$-free type σ, this yields $M \simeq_{\mathrm{obs}} \widehat{M}$, which achieves our purpose. $\square$

From here onwards, we shall therefore use the labels PCF, $\mathrm{T} + min$, W, etc. to refer to the $\times$-free versions of these languages, and shall only refer to types generated from $\mathtt{N}$ via $\to$.

2.2 Nested sequential procedures

Next, we summarize the necessary elements of the theory of *nested sequential procedures* (NSPs) also known as *PCF Böhm trees*,[6] relying on [20] for further details and for the relevant proofs. Although we shall provide enough of the formal details to support what we wish to do, a working intuition for NSPs is perhaps more easily acquired by looking at examples. The reader may therefore wish to look at the examples appearing from Definition 10 onwards in conjunction with the following definitions.

As explained in Subsection 3.2.5 and Section 6.1 of [20], *nested sequential procedures* (or *NSPs*) are infinitary terms generated by the following grammar, construed coinductively:

$$\begin{array}{rrcl} \textit{Procedures:} & p, q & ::= & \lambda x_0 \ldots x_{r-1}.\, e \\ \textit{Expressions:} & d, e & ::= & \bot \mid n \mid \mathtt{case}\ a\ \mathtt{of}\ (i \Rightarrow e_i \mid i \in \mathbb{N}) \\ \textit{Applications:} & a & ::= & x\, q_0 \ldots q_{r-1} \end{array}$$

Informally, an NSP captures the possible behaviours of a (sequential) program with inputs bound to the formal parameters x_j, which may themselves be of function type. Such a program may simply diverge ($\bot$), or return a value n, or apply one of its inputs

[6] The term 'nested sequential procedure' was adopted in [20] as a neutral label for a notion that is of equal relevance to both PCF and Kleene computability.

x_j to some arguments—the subsequent behaviour of the program may depend on the numerical result i of this call. Here the arguments to which x_j is applied are themselves specified via NSPs $q_0, \ldots, q_{r-1}$ (which may also involve calls to x_j). In this way, NSPs should be seen as syntax trees which may be infinitely deep as well as infinitely broad.

We use t as a meta-variable ranging over all three kinds of NSP terms. We shall often use vector notation $\vec{x}, \vec{q}$ for finite sequences $x_0 \ldots x_{r-1}$ and $q_0 \ldots q_{r-1}$. Note that such sequences may be empty, so that for instance we have procedures of the form $\lambda.e$. As a notational concession, we will sometimes denote the application of a variable x to an empty list of arguments by $x()$. If $p = \lambda\vec{y}.e$, we will also allow the notation $\lambda x.p$ to mean $\lambda x\vec{y}.e$. The notions of free variable and (infinitary) α-equivalence are defined in the expected way, and we shall work with terms only up to α-equivalence.

If variables x are considered as annotated with simple types σ (as indicated by writing x^σ), there is an evident notion of a *well-typed* procedure term $p : \sigma$, where $\sigma \in \mathsf{T}$. Specifically, within any term t, occurrences of procedures $\lambda\vec{x}.e$ (of any type), applications $x\vec{q}$ (of the ground type $\mathtt{N}$) and expressions e (of ground type) have types that are related to the types of their constituents and of variables as usual in typed λ-calculus extended by case expressions of type $\mathtt{N}$. We omit the formal definition here since everything works as expected; for a more precise formulation see [15, Section 6.1.1].

For each type σ, we let $\mathsf{SP}(\sigma)$ be the set of well-typed procedures of type σ, and $\mathsf{SP}^0(\sigma)$ for the set of *closed* such procedures; note that $\mathsf{SP}^0(\mathtt{N}) \cong \mathbb{N}_\perp$. As a notational liberty, we will sometimes write the procedures $\lambda.n, \lambda.\perp \in \mathsf{SP}^0(\mathtt{N})$ simply as $n, \perp$.

Note that in the language of NSPs, one cannot directly write e.g. $f^{\sigma\to\mathtt{N}}x^\sigma$, since the variable x^σ is not formally a procedure. However, we may obtain a procedure corresponding to x^σ via *hereditary η-expansion*: if $\sigma = \sigma_0 \to \cdots \to \sigma_{r-1} \to \mathtt{N}$, then we define a procedure $x^{\sigma\eta}$ inductively on types by

$$x^{\sigma\eta} \;=\; \lambda z_0^{\sigma_0} \cdots z_{r-1}^{\sigma_{r-1}}.\, \mathtt{case}\; x z_0^{\sigma_0\eta} \cdots z_{r-1}^{\sigma_{r-1}\eta}\; \mathtt{of}\; (i \Rightarrow i)\;.$$

For example, the procedure corresponding to the identity on type σ may now be written as $\lambda x^\sigma.x^{\sigma\eta}$. Note that $x^{\mathtt{N}\eta} = \lambda.\,\mathtt{case}\; x()\; \mathtt{of}\; (i \Rightarrow i)$.

In order to perform computation with NSPs, and in particular to define the *application* of a procedure p to an argument list $\vec{q}$, we shall use an extended calculus of *meta-terms*, within which the terms as defined above will play the role of normal forms. Meta-terms are generated by the following infinitary grammar, again construed coinductively:

$$\begin{array}{rrcl}
\textit{Meta-procedures:} & P, Q & ::= & \lambda\vec{x}.\, E \\
\textit{Meta-expressions:} & D, E & ::= & \perp \;\mid\; n \;\mid\; \mathtt{case}\; G\; \mathtt{of}\; (i \Rightarrow E_i \mid i \in \mathbb{N}) \\
\textit{Ground meta-terms:} & G & ::= & E \;\mid\; x\,\vec{Q} \;\mid\; P\vec{Q}
\end{array}$$

Again, our meta-terms will be subject to the evident typing rules which work as expected. Unlike terms, meta-terms are amenable to a notion of (infinitary) substitution: if T is a meta-term and $Q_0, \ldots, Q_{r-1}$ are meta-procedures whose types match those of $x_0, \ldots, x_{r-1}$ respectively, we have the evident meta-term $T[\vec{x} \mapsto \vec{Q}]$.

We equip our meta-terms with a *head reduction* $\rightsquigarrow_h$ generated as follows:

- $(\lambda\vec{x}.E)\vec{Q} \rightsquigarrow_h E[\vec{x} \mapsto \vec{Q}]$ (β-rule).

- $\texttt{case}\ \bot\ \texttt{of}\ (i \Rightarrow E_i) \rightsquigarrow_h \bot$.
- $\texttt{case}\ n\ \texttt{of}\ (i \Rightarrow E_i) \rightsquigarrow_h E_n$.
- $\texttt{case}\ (\texttt{case}\ G\ \texttt{of}\ (i \Rightarrow E_i))\ \texttt{of}\ (j \Rightarrow F_j) \rightsquigarrow_h$
  $\texttt{case}\ G\ \texttt{of}\ (i \Rightarrow \texttt{case}\ E_i\ \texttt{of}\ (j \Rightarrow F_j))$.
- If $G \rightsquigarrow_h G'$ and G is not a `case` meta-term, then

$$\texttt{case}\ G\ \texttt{of}\ (i \Rightarrow E_i) \rightsquigarrow_h \texttt{case}\ G'\ \texttt{of}\ (i \Rightarrow E_i)\ .$$

- If $E \rightsquigarrow_h E'$ then $\lambda\vec{x}.E \rightsquigarrow_h \lambda\vec{x}.E'$.

We write $\rightsquigarrow_h^*$ for the reflexive-transitive closure of $\rightsquigarrow_h$. We call a meta-term a *head normal form* if it cannot be further reduced using $\rightsquigarrow_h$. The possible shapes of meta-terms in head normal form are $\bot$, n, $\texttt{case}\ y\vec{Q}\ \texttt{of}\ (i \Rightarrow E_i)$ and $y\vec{Q}$, the first three optionally prefixed by $\lambda\vec{x}$.

We may now see how an arbitrary meta-term T may be evaluated to a normal form t, by a process analogous to the computation of Böhm trees in untyped λ-calculus. For the present paper a somewhat informal description will suffice; for a more formal treatment we refer to [20, Section 6.1]. First, we attempt to reduce T to head normal form by repeatedly applying head reductions. If a head normal form is never reached, the normal form is $\bot$ (possibly prefixed by some $\lambda\vec{x}$ appropriate to the type). If the head normal form is $\bot$ or n (possibly prefixed by $\lambda\vec{x}$), then this is the normal form of T. If the head normal form is $\texttt{case}\ y\vec{Q}\ \texttt{of}\ (i \Rightarrow E_i)$, then we recursively evaluate the Q_j and E_i to normal forms q_j, e_i by the same method, and take $t = \texttt{case}\ y\vec{q}\ \texttt{of}\ (i \Rightarrow e_i)$; likewise for `case` expressions prefixed by a λ, and for meta-terms $y\vec{Q}$. Since the resulting term t may be infinitely deep, this evaluation is in general an infinitary process in the course of which t crystallizes out, although any required finite portion of t may be computed by just finitely many reductions.

If $p = \lambda x_0 \cdots x_r.e \in \mathsf{SP}(\sigma \to \tau)$ and $q \in \mathsf{SP}(\sigma)$, we define the *application* $p{\cdot}q \in \mathsf{SP}(\tau)$ to be the normal form of the meta-procedure $\lambda x_1 \cdots x_r.\, e[x_0 \mapsto q]$. This makes the sets $\mathsf{SP}(\sigma)$ into a total applicative structure SP, and the sets $\mathsf{SP}^0(\sigma)$ of closed procedures into a total applicative structure SP^0.

We write $\sqsubseteq$ for the syntactic ordering on each $\mathsf{SP}(\sigma)$, so that $p \sqsubseteq p'$ if p is obtained from p' by replacing certain subterms (perhaps infinitely many) by $\bot$. It is not hard to check that each $\mathsf{SP}^0(\sigma)$ is a DCPO with this ordering, and that application is monotone and continuous with respect to this structure.

We also have an *extensional preorder* $\preceq$ on each $\mathsf{SP}^0(\sigma)$ defined as follows: if $p, p' \in \mathsf{SP}^0(\sigma)$ where $\sigma = \sigma_0 \to \cdots \to \sigma_{t-1} \to \mathtt{N}$, then

$$p \preceq p' \ \text{ iff } \ \forall q_0, \ldots, q_{t-1}.\forall n.\ (p \cdot q_0 \cdot \ldots \cdot q_{t-1} = \lambda.n) \ \Rightarrow\ (p' \cdot q_0 \cdot \ldots \cdot q_{t-1} = \lambda.n)$$

The following useful fact is established in Subsection 6.1.4 of [20]:

Theorem 6 (NSP context lemma) *If $p \preceq p' \in \mathsf{SP}^0(\sigma)$, then for all $r \in \mathsf{SP}^0(\sigma \to \mathtt{N})$ we have $r \cdot p \sqsubseteq r \cdot p'$.*

We now have everything we need to give an interpretation of simply typed λ-calculus in SP: a variable x^σ is interpreted by $x^{\sigma\eta}$, application is interpreted by $\cdot$, λ-abstraction is interpreted by itself, and we may also add a constant p for each $p \in \mathsf{SP}^0$ (interpreted by itself). The following non-trivial theorem, proved in [20, Section 6.1], ensures that many familiar kinds of reasoning work smoothly for NSPs:

Theorem 7 *SP^0 is a* typed $\lambda\eta$-algebra*: that is, if $U, V : \sigma$ are closed simply typed λ-terms with constants drawn from SP^0 and $U =_\beta V$, then U, V denote the same element of $\mathsf{SP}^0(\sigma)$ under the above interpretation.*

One of the mathematically interesting aspects of SP^0 is the existence of several well-behaved *substructures* corresponding to more restricted flavours of computation. The two substructures of relevance to this paper are defined as follows:

Definition 8 *(i) The* application tree *of an NSP term t is simply the tree of all occurrences of applications $x\vec{q}$ within t, ordered by subterm inclusion.*

(ii) An NSP term t is left-well-founded *(LWF) if its application tree is well-founded.*

(iii) A term t is left-bounded *if its application tree is of some finite depth d.*

The following facts are proved in [20, Section 6.3]:

Theorem 9 *(i) LWF procedures are closed under application. Moreover, the substructure $\mathsf{SP}^{0,\mathrm{lwf}}$ of SP^0 consisting of LWF procedures is a sub-$\lambda\eta$-algebra of SP^0.*

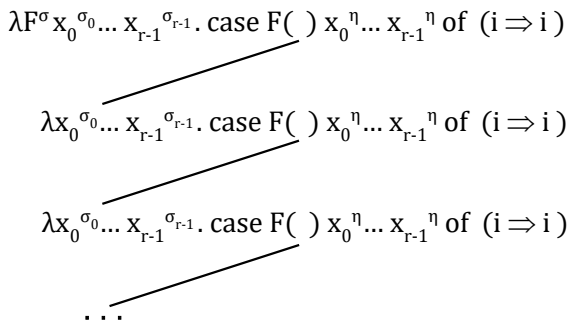
(ii) Left-bounded procedures are closed under application, and the corresponding substructure $\mathsf{SP}^{0,\mathrm{lbd}}$ is also a sub-$\lambda\eta$-algebra of SP^0.

Next, we indicate how NSPs provide us with a good model for the behaviour of terms of PCF + *byval*.

Definition 10 *To any* PCF + *byval term $M : \sigma$ (possibly with free variables) we may associate a procedure $[\![M]\!] \in \mathsf{SP}(\sigma)$ (with the same or fewer free variables) in a compositional way, by recursion on the term structure of M:*

- $[\![x^\sigma]\!] = x^{\sigma\eta}$.
- $[\![\lambda x.M]\!] = \lambda x.[\![M]\!]$.
- $[\![MN]\!] = [\![M]\!] \cdot [\![N]\!]$
- $[\![suc]\!] = \lambda x^{\mathtt{N}}.\,\mathtt{case}\ x()\ \mathtt{of}\ (0 \Rightarrow 1 \mid 1 \Rightarrow 2 \mid 2 \Rightarrow 3 \mid \cdots)$.
- $[\![pre]\!] = \lambda x^{\mathtt{N}}.\,\mathtt{case}\ x()\ \mathtt{of}\ (0 \Rightarrow 0 \mid 1 \Rightarrow 0 \mid 2 \Rightarrow 1 \mid \cdots)$.
- $[\![ifzero]\!] = \lambda x^{\mathtt{N}} y^{\mathtt{N}} z^{\mathtt{N}}.\,\mathtt{case}\ x()\ \mathtt{of}\ (0 \Rightarrow \mathtt{case}\ y()\ \mathtt{of}\ (j \Rightarrow j) \mid i+1 \Rightarrow \mathtt{case}\ z()\ \mathtt{of}\ (j \Rightarrow j))$.
- $[\![byval_\tau^{\vec{\sigma}}]\!] = \lambda f^{\vec{\sigma}\to\mathtt{N}\to\tau} \vec{x} n \vec{y}.\,\mathtt{case}\ n()\ \mathtt{of}\ (0 \Rightarrow \mathtt{case}\ f\vec{x}^{\,\eta}(\lambda.0)\vec{y}^{\,\eta}\ \mathtt{of}\ (j \Rightarrow j) \mid 1 \Rightarrow \mathtt{case}\ f\vec{x}^{\,\eta}(\lambda.1)\vec{y}^{\,\eta}\ \mathtt{of}\ (j \Rightarrow j) \mid \cdots)$.

- *If* $\sigma = \sigma_0 \to \cdots \sigma_{r-1} \to \mathtt{N}$*, then* $[\![Y_\sigma]\!]$ *is the NSP depicted below:*



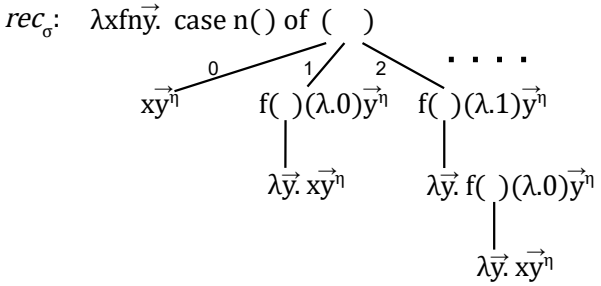
The NSP for Y_σ is the archetypal example of a non-LWF procedure: the nested sequence of application subterms $F(\cdots)$ never bottoms out.

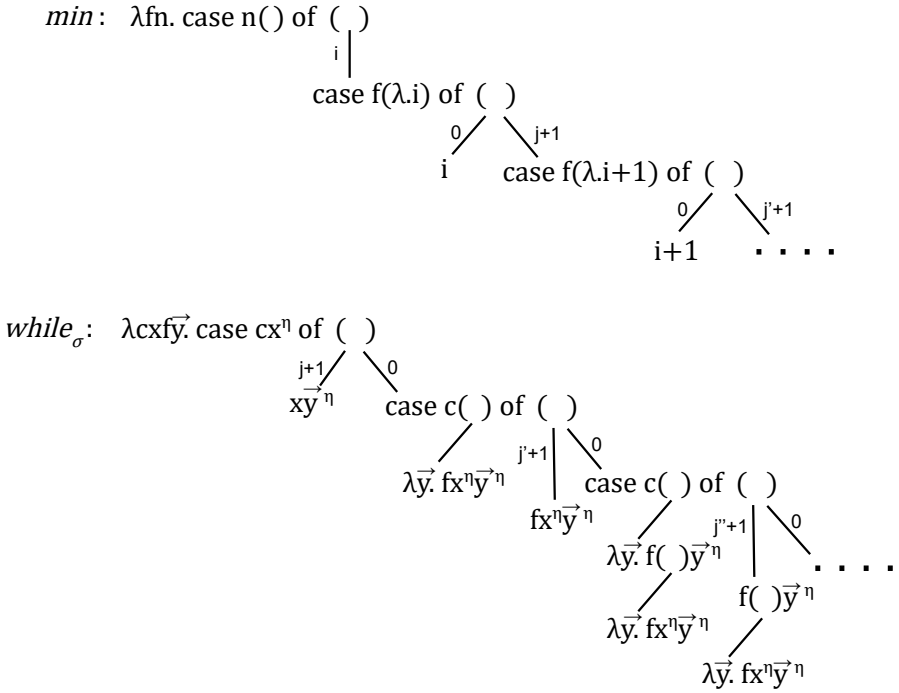
The following theorem, proved in [20, Subsection 7.1.3], confirms that this interpretation is faithful to the behaviour of PCF + *byval* programs:

Theorem 11 (Adequacy of NSP model) *If* M *is any closed* PCF + *byval term of type* $\mathtt{N}$*, then* $M \rightsquigarrow^* n$ *iff* $[\![M]\!] = \lambda.n$*, and* M *diverges iff* $[\![M]\!] = \lambda.\bot$*.*

It is also the case that every *computable* element of any $\mathsf{SP}^0(\sigma)$ is denotable by a closed PCF + *byval* term of type σ (see [20, Subsection 7.1.5]), though we shall not need this fact in this paper.

We also obtain interpretations of T + *min* and W in SP^0, induced by the translations of these languages into PCF as described in Subsection 2.1. Applying the definition of $[\![-]\!]$ above to the PCF programs Rec_σ, *Min*, $While_\sigma$, we may thus obtain the appropriate NSPs for the constants rec_σ, *min*, $while_\sigma$ respectively. To reduce clutter, we allow an application term a to stand for the expression $\mathtt{case}\ a\ \mathtt{of}\ (i \Rightarrow i)$. Where a case branch label involves a metavariable i or j, there is intended to be a subtree of the form displayed for each $i, j \in \mathbb{N}$.





Clearly, each of these NSPs is left-well-founded; the one for *min* is even left-bounded, since there is no nesting of calls to n or f. Since also the NSPs for *suc*, *pre*, *ifzero*, *byval* and $x^{\sigma\eta}$ are plainly LWF, and LWF procedures are closed under λ-abstraction and application (Theorem 9(i)), we have the following important result:

Theorem 12 *The interpretation of any term of* $\mathrm{T} + min$ *or* W *is an LWF procedure.*

The reader may also enjoy constructing the appropriate trees for rec_σ^{str} and $while_\sigma^{str}$ where σ is of level 0, and to observe that these trees are *left-bounded* (in contrast to those for rec_σ and $while_\sigma$). Since the trees for *min* and all the constants of B are also left-bounded, we may infer by Theorem 9(ii) that the interpretation of any term of $\mathrm{T}_0^{str} + min$ or W_0^{str} is left-bounded. This may shed light on the discussion of Section 1, but will not be formally required for the remainder of the paper.

2.3 Bar recursors

We conclude the section by explaining the notions of bar recursion that we shall use. Traditionally, bar recursion has usually been considered either as a purely syntactic operation (as in [27]), or as an element of a total type structure such as the Kleene-Kreisel *total continuous* functionals Ct or Bezem's *strongly majorizable* functionals SM. Here, for expository purposes, we shall introduce bar recursion first in a 'naive' way with reference to the full set-theoretic type structure S, and then as an operation within SP^0 defined by a certain PCF program—the latter will provide the setup for the main

theorem of Section 3. In Section 4, we will relate this to more familiar notions of bar recursion and will show how our results transfer relatively easily from SP^0 to models such as Ct and SM.

As explained in Section 1, bar recursion is in essence recursion over well-founded trees. For the purpose of this paper, a *tree* $\mathcal{T}$ will be an inhabited prefix-closed subset of $\mathbb{N}^*$ (the set of finite sequences over $\mathbb{N}$), with the property that for every $\vec{x} = (x_0, \ldots, x_{i-1}) \in \mathcal{T}$, one of the following holds:

1. There is no $y \in \mathbb{N}$ such that $(x_0, \ldots, x_{i-1}, y) \in \mathcal{T}$ (we then say $\vec{x}$ is a *leaf* of $\mathcal{T}$).
2. For all $y \in \mathbb{N}$, $(x_0, \ldots, x_{i-1}, y) \in \mathcal{T}$ (we then say $\vec{x}$ is an *internal node* of $\mathcal{T}$).

We write $\mathcal{T}^l, \mathcal{T}^n$ for the set of leaves and internal nodes of $\mathcal{T}$ respectively.

A tree $\mathcal{T}$ is *well-founded* if there is no infinite sequence $x_0, x_1, \ldots$ over $\mathbb{N}$ such that $(x_0, \ldots, x_{i-1}) \in \mathcal{T}$ for every i. Thus, in a well-founded tree, every maximal path terminates in a leaf. If $\mathcal{T}$ is well-founded, a function $f : \mathcal{T} \to \mathbb{N}$ may be defined by recursion on the tree structure if we are given the following data:

- A *leaf function* $L : \mathcal{T}^l \to \mathbb{N}$ specifying the value of f on leaf nodes.
- A *branch function* $G : \mathcal{T}^n \times \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ specifying the value of f on an internal node $\vec{x}$, assuming we have already defined the value of f on all the immediate children of f. Specifically, if $b(y)$ gives the value of $f(\vec{x}, y)$ for every $y \in \mathbb{N}$, then $G(\vec{x}, b)$ gives the value of $f(\vec{x})$.

Indeed, we may define the function $BR^{\mathcal{T}}_{L,G}$ obtained by *bar recursion* from L and G to be the unique function $\mathcal{T} \to \mathbb{N}$ satisfying:

$$\begin{array}{lcll} BR^{\mathcal{T}}_{L,G}(\vec{x}) & = & L(\vec{x}) & \text{if } \vec{x} \in \mathcal{T}^l \\ BR^{\mathcal{T}}_{L,G}(\vec{x}) & = & G(\vec{x},\ \Lambda y.\, BR^{\mathcal{T}}_{L,G}(\vec{x}, y)) & \text{if } \vec{x} \in \mathcal{T}^n \end{array}$$

The existence and uniqueness of $BR^{\mathcal{T}}_{L,G}$ are easy consequences of well-foundedness.

It is easy to see how L and G may be represented by objects of simple type: elements of $\mathcal{T}$ may be represented by elements of $\mathbb{N}$ via some standard primitive recursive coding $\langle \cdots \rangle : \mathbb{N}^* \to \mathbb{N}$, so that we may consider L and G as functions $\mathbb{N} \to \mathbb{N}$ and $\mathbb{N} \times \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ respectively. As regards the tree $\mathcal{T}$ itself, we now introduce two related ways, due respectively to Spector [27] and Kohlenbach [13], for representing well-founded trees by means of certain functionals $F : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$. We shall write $|\vec{x}|$ for the length of a sequence $\vec{x}$, and if $j \in \mathbb{N}$, shall write $[\vec{x}\, j^\omega]$ for the primitive recursive function $\mathbb{N} \to \mathbb{N}$ defined by

$$[x_0, \ldots, x_{r-1}, j^\omega](i) \ = \ \begin{cases} x_i & \text{if } i < r, \\ j & \text{if } i \geq r \end{cases}$$

Definition 13 *Suppose F is any function $\mathbb{N}^{\mathbb{N}} \to \mathbb{N}$.*

(i) We say $\vec{x} \in \mathbb{N}^*$ satisfies the Spector bar condition *(with respect to F) if*

$$F([\vec{x}\, 0^\omega]) \ < \ |\vec{x}|\,,$$

and the Kohlenbach bar condition *if*

$$F([\vec{x}\,0^\omega]) \;=\; F([\vec{x}\,1^\omega])\;.$$

(ii) The Spector tree *of F, written $\mathcal{T}^S(F)$, is the set of sequences $\vec{x} \in \mathbb{N}^*$ such that no proper prefix of $\vec{x}$ satisfies the Spector bar condition w.r.t. F. The* Kohlenbach tree *$\mathcal{T}^K(F)$ is defined analogously using the Kohlenbach bar condition.*

Both $\mathcal{T}^S(F)$ and $\mathcal{T}^K(F)$ are clearly trees in our sense. Furthermore, the following important fact ensures a plentiful supply of functionals giving rise to *well-founded* trees.

Proposition 14 *If F is continuous with respect to the usual Baire topology on $\mathbb{N}^\mathbb{N}$, then both $\mathcal{T}^S(F)$ and $\mathcal{T}^K(F)$ are well-founded.*

PROOF SKETCH: For any infinite sequence $x_0, x_1, \ldots$, there will be some 'modulus of continuity' m for F such that for all $n \geq m$ and all j we have $F([x_0, \ldots, x_{n-1}, j^\omega]) = F(\Lambda i.x_i)$. It follows easily that some finite subsequence $(x_0, \ldots, x_{n-1})$ will satisfy the Spector [resp. Kohlenbach] condition. The shortest such prefix will then be a leaf in $\mathcal{T}^S(F)$ [resp. $\mathcal{T}^K(F)$]. □

There are also other ways in which well-founded trees may arise—for instance, it is well-known that if F is *majorizable* then $\mathcal{T}^S(F)$ is well-founded—but it is the continuous case that will be most relevant to our purposes.

Using the above representations, we may now introduce our basic definition of bar recursion. We are here naively supposing that F, L, G are drawn from the full set-theoretic type structure S, although this is not really the typical situation. Relativizations of this definition to other total type structures will be considered in Section 4.

Definition 15 (Bar recursors) *A* Spector bar recursor *(over S) is any partial function*

$$BR \;:\; \mathsf{S}((\mathtt{N} \to \mathtt{N}) \to \mathtt{N}) \;\times\; \mathsf{S}(\mathtt{N} \to \mathtt{N}) \;\times\; \mathsf{S}(\mathtt{N} \times (\mathtt{N} \to \mathtt{N}) \to \mathtt{N}) \;\rightharpoonup\; \mathbb{N}$$

such that for all $F : \mathbb{N}^\mathbb{N} \to \mathbb{N}$ with $\mathcal{T}^S(F)$ well-founded and for any $L : \mathbb{N} \to \mathbb{N}$ and $G : \mathbb{N} \times \mathbb{N}^\mathbb{N} \to \mathbb{N}$, we have

$$\begin{aligned} BR(F,L,G)(\langle\vec{x}\rangle) &= L(\langle\vec{x}\rangle) \quad \textit{whenever } \vec{x} \in \mathcal{T}^S(F)^l, \\ BR(F,L,G)(\langle\vec{x}\rangle) &= G(\langle\vec{x}\rangle,\ \Lambda z.\, BR(F,L,G)(\langle\vec{x},z\rangle) \quad \textit{whenever } \vec{x} \in \mathcal{T}^S(F)^n. \end{aligned}$$

The notion of Kohlenbach bar recursor is defined analogously using $\mathcal{T}^K(F)$.

Note that the above equations uniquely fix the value of $BR(F,L,G)(\langle\vec{x}\rangle)$ for all $\vec{x} \in \mathcal{T}^S(F)$. We shall not be concerned with the behaviour of bar recursors BR on sequences $\vec{x}$ outside the tree in question,[7] nor with their behaviour on arguments F such that $\mathcal{T}(F)$ is not well-founded.

Our next step will be to re-construe the definition of bar recursors as a recursive program within PCF. For this, we first note that both the Spector and Kohlenbach

[7] In this respect our definition of 'bar recursor' here is slightly weaker than some given in the literature (e.g. in [20, Section 7.3]); this will in principle make our main theorem slightly stronger, though not in any deep or essential way.

bar conditions are readily testable by means of programs in PCF or indeed in T_0^{str}. Although these languages have just a single base type $\mathtt{N}$, for clarity we shall write $\mathtt{N}^*$ for occurrences of $\mathtt{N}$ whose role is to represent sequences $\vec{x}$ via their codes $\langle\vec{x}\rangle$. We shall suppose we have a fixed choice of T_0^{str} programs

$$\mathrm{len} \;:\; \mathbb{N}^* \to \mathbb{N}\,, \qquad \mathrm{add} \;:\; \mathbb{N}^* \to \mathbb{N} \to \mathbb{N}^*\,, \qquad \mathrm{basic} \;:\; \mathbb{N}^* \to \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

such that for any $\vec{x}, z, j, i$ we have

$$\mathrm{len}\,\langle\vec{x}\rangle \;\rightsquigarrow^*\; |\vec{x}|\,, \qquad \mathrm{add}\,\langle\vec{x}\rangle\; z \;\rightsquigarrow^*\; \langle\vec{x}, z\rangle\,, \qquad \mathrm{basic}\;\vec{x}\; j\; i \;\rightsquigarrow^*\; [\vec{x}, j](i)\,,$$

(where we omit the hats from PCF numerals). We also presuppose fixed T_0^{str} implementations of $=$ and $<$. Using this machinery, we may now define a PCF term

$$\mathsf{BR}^S \;:\; ((\mathtt{N} \to \mathtt{N}) \to \mathtt{N}) \;\to\; (\mathtt{N}^* \to \mathtt{N}) \;\to\; (\mathtt{N}^* \to (\mathtt{N} \to \mathtt{N}) \to \mathtt{N}) \;\to\; (\mathtt{N}^* \to \mathtt{N})$$

by

$$\begin{array}{rcl} \mathsf{BR}^S\; F\; L\; G\; x & = & \mathit{if}\;\; (F(\mathrm{basic}(x,0)) < \mathrm{len}\; x)\;\; \mathit{then}\;\; L\; x \\ & & \mathit{else}\;\; G\; x\; (\lambda z.\, \mathsf{BR}^S\; F\; L\; G\; (\mathrm{add}\; x\; z)) \end{array}$$

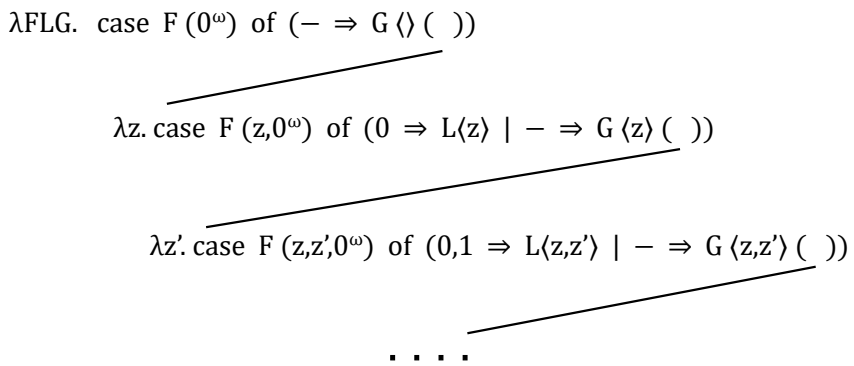
or a little more formally by

$$\begin{array}{rcl} \mathsf{BR}^S & = & \lambda FLG.\; Y_{\mathtt{N}\to\mathtt{N}}\,(\lambda B.\; \lambda x. \\ & & \qquad \mathit{if}\;\; (F(\mathrm{basic}(x,0)) < \mathrm{len}\; x)\;\; \mathit{then}\;\; L\; x\;\; \mathit{else}\;\; G\; x\; (\lambda z.\, B(\mathrm{add}\; x\; z)))\,. \end{array}$$

The Kohlenbach version BR^K (of the same type) is defined analogously, replacing the subterm $(F(\mathrm{basic}(x,0)) < \mathrm{len}\; x)$ by $(F(\mathrm{basic}(x,0)) = F(\mathrm{basic}(x,1)))$.

Both of these PCF terms may be interpreted in SP^0, yielding NSPs at the above type which we shall also denote by BR^S and BR^K respectively. We shall refer to these as the *canonical (Spector or Kohlenbach) bar recursors* within SP^0.

It is not hard to see intuitively that these NSPs for BR^S and BR^K are non-LWF, since the unrolling of the recursion will lead to an infinite sequence of nested calls to G. To illustrate the phenomenon, we schematically depict here part of the NSP for $\lambda FLG.\, \mathsf{BR}^S\, F\, L\, G\, \langle\,\rangle$ (so that evaluations of x are elided):



It follows immediately by Theorem 12 that these particular bar recursors within SP^0 are not denotable in $\mathrm{T}+\mathit{min}$. However, this does not in itself address the main question of interest: what we wish to know is that no element of SP^0 can have the extensional behaviour of a bar recursor, even if we restrict attention to arguments F, L, G which represent *total* functionals in the spirit of Definition 15. This begs the question of what it means for an element of SP^0 to 'represent' a total functional. We now briefly indicate why there is room for several reasonable answers to this question, thus motivating our 'robust' approach which is designed to work for all of them.

The general picture we have in mind is that of some chosen type structure T of total functionals over $\mathbb{N}$—that is, a family of sets $T(\sigma)$ where $T(\mathtt{N}) = \mathbb{N}$ and $T(\sigma \to \tau)$ is some set of total functions $T(\sigma) \to T(\tau)$—along with some way of representing T within SP^0. The latter will in general consist of what in [20] we call a (type- and numeral-respecting) *applicative simulation* $\gamma : T \mathrel{—\!\!\triangleright} \mathsf{SP}^0$: that is, a family of total relations $\gamma_\sigma \subseteq T(\sigma) \times \mathsf{SP}^0(\sigma)$ such that $\gamma_N(n, x)$ iff $x = n$, and $\gamma_{\sigma\to\tau}(f, f')$ and $\gamma_\sigma(x, x')$ imply $\gamma(f(x), f' \cdot x')$. Relative to this, we may call an element of $\mathsf{SP}^0(\sigma)$ *total* if it is in the image of γ_σ. We might then regard as a 'bar recursor' in SP^0 any procedure Φ that satisfies (an analogue of) the equations of Definition 15 for all total $F, L, G \in \mathsf{SP}^0$ of appropriate types.

However, there is scope for variation here, both in the choice of T (which could be either Ct or HEO, for example) and in the choice of the simulation γ. Different choices will in general lead to different notions of 'total element' within SP^0 (this phenomenon is explored in [24]), and hence to different criteria for what it means to be a bar recursor in SP^0.

Our approach to dealing with this is to identify a core class of elements of SP^0 which are likely to be 'total' under all reasonable choices of interest, and then postulate, as a minimal requirement for any proposed 'bar recursor' in SP^0, that the equations of Definition 15 should at least be satisfied by these core total elements. Our main theorem will then claim that even this minimal requirement cannot be met by any LWF procedure. As we shall argue in Section 4, this will enable us to conclude that in no reasonable sense can a bar recursor be definable in $\mathrm{T} + \mathit{min}$ or W.

In fact, a suitable class of core total elements for our purpose will be those definable in the language T_0^{str} of Subsection 2.1.1 (interpreted in SP^0 via its translation to PCF). Our rationale for this is that all reasonable choices of the total type structure T can be expected to be models for T_0^{str} (or equivalently for Kleene's S1–S8), and that it is furthermore a mild requirement that γ should relate the interpretation of any closed T_0^{str} term in T to its interpretation in SP^0. (This will in fact be so if it is the case for the standard programs k and s and for the constants *suc*, *pre*, *ifzero*, $\mathit{rec}_\sigma^{str}$.) Our thesis, then, is that the T_0^{str}-definable elements of SP^0 can be expected to be 'total' in all senses of interest.

All of this leads us to the following definitions. Here we identify the PCF programs basic, add, len with their interpretations in SP^0.

Definition 16 *Suppose* $F \in \mathsf{SP}^0((\mathtt{N} \to \mathtt{N}) \to \mathtt{N})$

(i) We say $\vec{x} \in \mathbb{N}^*$ *satisfies the Spector bar condition with respect to* F *if* $F \cdot (\mathrm{basic} \cdot \langle\vec{x}\rangle \cdot 0) < \mathrm{len} \cdot \langle\vec{x}\rangle$, *and the Kohlenbach bar condition if* $F \cdot (\mathrm{basic} \cdot \langle\vec{x}\rangle \cdot 0) = F \cdot (\mathrm{basic} \cdot \langle\vec{x}\rangle \cdot 1)$.

(ii) The tree $\mathcal{T}^S(F)$ [resp. $\mathcal{T}^K(F)$] consists of all sequences $\vec{x}$ such that no proper prefix of $\vec{x}$ satisfies the Spector [resp. Kohlenbach] bar condition.

The following easy fact will be useful:

Proposition 17 *If $F \in \mathsf{SP}^0((\mathtt{N} \to \mathtt{N}) \to \mathtt{N})$ is definable by a term of T_0^{str}, then $\mathcal{T}^S(F)$, $\mathcal{T}^K(F)$ are well-founded.*

Proof: If F is T_0^{str}-definable, clearly F will represent a total functional $\overline{F} : \mathbb{N}^{\mathbb{N}} \to \mathbb{N}$ with respect to the obvious representation of functions $\mathbb{N} \to \mathbb{N}$ within $\mathsf{SP}^0(\mathtt{N} \to \mathtt{N})$. Moreover, since application in SP^0 is continuous, it is easy to see that $\overline{F}$ will be continuous for the Baire topology, and so by Proposition 14 the trees $\mathcal{T}^S(F) = \mathcal{T}^S(\overline{F})$ and $\mathcal{T}^K(F) = \mathcal{T}^K(\overline{F})$ will be well-founded. □

Definition 18 *A* weak Spector bar recursor *in SP^0 is an element*

$$\Phi \ \in \ \mathsf{SP}^0 \ (((\mathtt{N} \to \mathtt{N}) \to \mathtt{N}) \ \to \ (\mathtt{N}^* \to \mathtt{N}) \ \to \ (\mathtt{N}^* \to (\mathtt{N} \to \mathtt{N}) \to \mathtt{N}) \ \to \ (\mathtt{N}^* \to \mathtt{N}))$$

such that the following hold for all T_0^{str}-definable $F, L, G \in \mathsf{SP}^0$ of appropriate types such that for all $\vec{x} \in \mathcal{T}^S(F)$:

$$\begin{array}{lcl} \Phi \cdot F \cdot L \cdot G \cdot \langle\vec{x}\rangle & = & L \cdot \langle\vec{x}\rangle \quad \text{if } \vec{x} \in \mathcal{T}^S(F)^l \ , \\ \Phi \cdot F \cdot L \cdot G \cdot \langle\vec{x}\rangle & = & G \cdot \langle\vec{x}\rangle \cdot (\lambda z^{\mathtt{N}}.\, \Phi \cdot F \cdot L \cdot G \cdot (\mathrm{add} \cdot \langle\vec{x}\rangle \cdot z)) \quad \text{if } \vec{x} \in \mathcal{T}^S(F)^n \ . \end{array}$$

The notion of weak Kohlenbach bar recursor *in SP^0 is defined analogously.*

Here the abstraction λz is understood simply as a λ-abstraction within the language of NSPs; note that for any given $\Phi, F, L, G, \vec{x}$, the body of this abstraction will evaluate to an NSP with free variable z.

Clearly the canonical bar recursors BR^S, BR^K defined earlier are examples of weak bar recursors in this sense. Moreover:

Proposition 19 *If Φ is any weak (Spector or Kohlenbach) bar recursor, F, L, G are T_0^{str}-definable, and $\vec{x} \in \mathcal{T}(F)$, then the value of $\Phi \cdot F \cdot L \cdot G \cdot \langle\vec{x}\rangle$ is a numeral and is uniquely determined by the defining equations above.*

Proof: For a given F, L, G, we show that the set S of $\vec{x} \in \mathcal{T}(F)$ for which the proposition holds contains all leaves and all internal nodes whose immediate children are all in S; it follows that S is the whole of $\mathcal{T}(F)$ since the latter is well-founded. For the step case, we use the fact that if G is T_0^{str}-definable and $h \in \mathsf{SP}^0(\mathtt{N} \to \mathtt{N})$ is total (i.e. $h \cdot z \in \mathbb{N}$ for all $z \in \mathbb{N}$), then $G \cdot x \cdot h \in \mathbb{N}$ for any $x \in \mathbb{N}$. We show this by an easy induction on the T_0^{str} term that denotes G, which we may assume to be some β-normal form $\lambda x h.M$, so that all variables are bound within M are of type level 0. □

Clearly, if F represents an element $\check{F} \in \mathsf{Ct}((\mathtt{N} \to \mathtt{N}) \to \mathtt{N})$ via any reasonable simulation γ, it will be automatic that $\mathcal{T}(F)$ is well-founded since any such $\check{F}$ is continuous. Indeed, since bar recursors exist as third-order functionals $\check{\Phi}$ within Ct, any elements $\Phi \in \mathsf{SP}^0$ that represent such $\check{\Phi}$ will be *total* weak bar recursors relative to γ. The

situation is different for the type structure HEO: there are classically discontinuous functions $\check{F} \in \mathsf{HEO}((\mathtt{N} \to \mathtt{N}) \to \mathtt{N})$, and if $F \in \mathsf{SP}^0$ represents such an $\check{F}$ then $\mathcal{T}(F)$ may be non-well-founded, in which case Definition 18 places no condition on how Φ should behave on F.

In our main proof, we shall find it more convenient to work with the Kohlenbach definition, but the theorem will transfer readily to the Spector version in view of the following easy relative definability result. From here on, we shall allow ourselves to write k for the pure type of level k, so that 0 denotes $\mathtt{N}$ and $k+1$ denotes $k \to \mathtt{N}$.

Proposition 20 *If Φ^S is any weak Spector bar recursor in SP^0, then a weak Kohlenbach bar recursor Φ^K is $\mathrm{T} + min$ definable relative to Φ^S. Hence if an LWF weak Spector bar recursor exists in SP^0, then so does an LWF weak Kohlenbach bar recursor.*

Proof: We first construct a $\mathrm{T} + min$ definable element $U \in \mathsf{SP}^0(2 \to 2)$ such that for any $F \in \mathsf{SP}^0(2)$ whose Kohlenbach tree is well-founded and is not simply $\{\langle\rangle\}$, the Spector tree of $U \cdot F$ is precisely the Kohlenbach tree of F. We may achieve this by defining (in PCF-style notation)

$$U \;=\; \lambda F.\lambda g.\; (min\ r.\ F([g(0),\ldots,g(r-1),0^\omega]) = F([g(0),\ldots,g(r-1),1^\omega])) - 1\ .$$

Using this, we may define

$$\Phi^K \;=\; \lambda FLGx.\ if\ F([0^\omega]) = F([1^\omega])\ then\ L\langle\rangle\ else\ \Phi^S(U(F),L,G)(x)\ .$$

Since $\mathcal{T}^K(F) = \mathcal{T}^S(U \cdot F)$ except in the trivial case, the equations stating that Φ^S is a weak Spector bar recursor (as in Definition 18) now readily imply the corresponding equations stating that Φ^K is a weak Kohlenbach bar recursor. $\square$

Conversely, a more subtle argument (given in Kohlenbach [13]) shows that Spector bar recursion is definable from Kohlenbach bar recursion even in System T, though we shall not need this here. We also refer the reader to Escardó and Oliva [8] for a cornucopia of related functionals known to be either interdefinable with or stronger than Spector bar recursion over System T; our main theorem will thus yield that none of these functionals are definable in $\mathrm{T} + min$.

One final preliminary is needed. In order to ease notation in our main proof, we shall actually consider a simpler kind of bar recursor readily obtained as a specialization of those described above.

Definition 21 *A* simplified weak Spector bar recursor *(in SP^0) is an element*

$$\Phi \in \mathsf{SP}^0(2 \to 2 \to 1)$$

such that the following hold for all T_0^{str}-definable $F, G \in \mathsf{SP}^0$ of appropriate types such that for all $\vec{x} \in \mathcal{T}^S(F)$:

$$\begin{array}{lcl} \Phi \cdot F \cdot G \cdot \langle\vec{x}\rangle & = & 2\langle\vec{x}\rangle + 1 \quad if\ \vec{x} \in \mathcal{T}^S(F)^l\ , \\ \Phi \cdot F \cdot G \cdot \langle\vec{x}\rangle & = & G \cdot (\lambda z^{\mathtt{N}}.\ \Phi \cdot F \cdot G \cdot (\mathrm{add} \cdot \vec{x}\ \cdot z)) \quad if\ \vec{x} \in \mathcal{T}^S(F)^n\ . \end{array}$$

The notion of simplified weak Kohlenbach bar recursor *is defined analogously.*

It is easily seen that a simplified weak (Spector or Kohlenbach) bar recursor is T_0^{str}-definable from an ordinary one just by specializing the leaf function L to $\lambda x.2x+1$ (this move is admittedly hard to motivate at this point!) and by eschewing the dependence of G on an argument x. By analogy with Proposition 19, we have:

Proposition 22 *If Φ is any simplified weak (Spector or Kohlenbach) bar recursor, F, G are T_0^{str}-definable, $\mathcal{T}(F)$ is well-founded and $\vec{x} \in \mathcal{T}(F)$, then the value of $\Phi \cdot F \cdot G \cdot \langle \vec{x} \rangle$ is a numeral and is uniquely determined by the defining equations above.*

The proof of Proposition 20 clearly also yields the following:

Proposition 23 *If an LWF simplified weak Spector bar recursor exists in SP^0, so does an LWF simplified weak Kohlenbach bar recursor.*

Such simplified bar recursors were called *restricted* bar recursors in [20, Section 6.3.4], but the latter name clashes with a different use of the same term by Spector in [27]. In [20] we supposed that the simplified bar recursors were weaker than the general ones, so that they led to a slightly stronger non-definability result. Actually, it turns out to be not too hard to define general bar recursors from simplified ones (we leave this as an exercise for the interested reader). Nevertheless, we shall prove our main theorem for the simplified versions, both because that was what was claimed in [20], and because it does lighten the notational load in parts of our proof. Against this, the later parts of the proof (Sections 3.5 and 3.6) turn out to be a little more delicate in the simplified setting, but we think there is also some interest in the opportunity this gives for illustrating the versatility of our method of proof.

3 The main theorem

In this section, we shall prove the following theorem:

Theorem 24 Within SP^0, no simplified weak Kohlenbach bar recursor can be LWF, and hence none can be definable in $\mathrm{T} + \mathit{min}$ or in W.

The corresponding fact for simplified Spector bar recursion (stated as Theorem 6.3.28 in [20]) will then follow immediately by Proposition 23. It will also follow, *a fortiori*, that no ordinary Spector or Kohlenbach bar recursor in the sense of Definition 15 can be LWF. From here on we shall consider only Kohlenbach bar recursion, and will write $\mathcal{T}^K(F)$ simply as $\mathcal{T}(F)$.

The proof of Theorem 24 follows the method of proof of Theorem 6.3.27 in [20], which shows that no NSP weakly representing the System T recursor $rec_{\mathtt{N}\to\mathtt{N}}$ can be definable in $\mathrm{T}_0^{str} + \mathit{min}$. The proof of this theorem is already quite intricate, and that of the present theorem adds some further ingredients. The reader may therefore find it helpful to study the proof of [20, Theorem 6.3.27] in conjunction with the present one—however, the account given here will be technically self-contained, and we shall also offer a more extended motivational discussion here than we did in [20].[8]

[8] We take the opportunity to draw attention here to a small error in the proof of [20, Theorem 6.3.27]. In the penultimate sentence of the proof (on page 258), we claim that $F_1(g^{F_1}_{()}) = K$. For this, we use

We start with an informal outline of our argument. Suppose that Φ is any genuine (simplified, weak) bar recursor as per Definition 21, and that Ψ is some LWF procedure purported to be such a bar recursor. We shall set $\Phi_0 = \lambda FG.\, \Phi FG\langle\rangle$ and $\Psi_0 = \lambda FG.\, \Psi FG\langle\rangle$, so that Ψ_0 is also LWF by Theorem 9(i). Our task will be to find some particular T_0^{str}-definable arguments $F \in \mathsf{SP}^0(2)$, $G \in \mathsf{SP}^0(2)$ such that $\Psi_0 \cdot F \cdot G \neq \Phi_0 \cdot F \cdot G$. Since Φ here is an arbitrary bar recursor, this will show that Ψ is not a bar recursor after all.

The key idea is that Ψ, being LWF, will only be prepared to nest calls to G to a finite depth along any specified computation path. On the other hand, Φ, being a genuine bar recursor, must be willing to nest such calls to any depth required, as dictated by $\mathcal{T}(F)$. In order to manifest an extensional difference between Φ_0 and Ψ_0, we therefore wish to construct an F such that $\mathcal{T}(F)$ that goes deeper on some path than Ψ_0 is willing to explore, together with a G that forces the computation of Φ_0 to explore precisely this path. In this way, we can arrange that the computation of $\Phi_0 \cdot F \cdot G$ retrieves from within G some numerical value K that is not discoverable by Ψ_0, and propagates it to the top level.

Much of the proof is aimed at acquiring a sufficient grasp of the behaviour of Ψ_0 that we can guarantee that $\Psi_0 \cdot F \cdot G$ does *not* return this value K. Our approach to this will be similar to that in [20, Theorem 6.3.27]. Assuming for the moment that we know how to obtain a suitable F, we shall start by considering the computation of $\Psi_0 \cdot F \cdot G_0$ for a certain very simple functional G_0. Suppose that this evaluates to some number c. By analysing this and some related computations in detail, we shall discover a set of properties of G_0 that suffice to *secure* this computation, in the sense that for any other G with these properties, a precisely similar computation will go through, yielding the same result c. Put another way, we shall find a certain *neighbourhood* $\mathcal{G}$ of G_0 such that for all $G \in \mathcal{G}$ we have $\Psi_0 \cdot F \cdot G = c$. Moreover, the construction of $\mathcal{G}$ will be so arranged that it is possible to pick some $G_1 \in \mathcal{G}$ which forces Φ_0 to explore beyond the reach of Ψ_0 in the manner suggested above. Indeed, by choosing such a G_1 with some care, we can ensure that $\Phi_0 \cdot F \cdot G_1$ evaluates to a number K chosen to be different from c. This establishes the required difference between Φ_0 and Ψ_0.

The main new ingredient, not present in the proof of [20, Theorem 6.3.27], concerns the way in which a suitable argument F is chosen. As indicated above, we want F to represent a well-founded tree that 'undercuts' the tree explored by Ψ_0 in a certain computation; on the other hand, the computation performed by Ψ_0 will itself depend partly on the argument F that we give it. This apparent circularity suggests that we should try to arrive at a suitable F (which we call F_∞) by a process of successive

---

the previously established fact that $g_{()}^{F_1}(0) = \langle 0, \ldots, 0\rangle$; in the context of the proof, it is natural to denote this by y_0. However, one also wants that y_0 is distinct from all of $y_1, \ldots, y_d$, which we do not here know to be the case.

The problem is readily fixed by simply adding, at the point at which each y_w is selected (for $1 \leq w \leq d$), the further requirement that $y_w \neq y_0$. For this, we need to add 1 to the lower bounds on the moduli m^w from earlier in the proof, so that we take $m^0 > n^0 + 2$, $m^1 > n^0 + n^1 + 3$, etc. Then, when picking the path through the tree for Ψ_d at the bottom of page 256, we should start by defining $y_0 = g_{()}^{F_0}(0)$ (so that actually $y_0 = c$), then insert the requirements that $y_1, y_2, \ldots$ differ from y_0. (In the last line of page 256, y_0 was originally intended to read y_1, but should now be modified to y_0, y_1.)

In the third-to-last line of page 257, the claim that $g_{(\vec{z},0)}^{F_1}(0) = y_w$ now holds even when $w = 0$.

approximation in tandem with our analysis of the computation of $\Psi_0 \cdot F \cdot G_0$. This will allow us to ensure that $\mathcal{T}(F_\infty)$ undercuts Ψ_0 with respect to the computation of $\Psi_0 \cdot F_\infty \cdot G_0$ itself.

More specifically, our proof will be structured as follows. In Section 3.1 we begin with some very simple functionals $G_0 \in \mathsf{SP}^0(2)$ and $F_0^+ \in \mathsf{SP}^0(2)$, of which the latter will serve as the first step in the iterative construction of a suitable F. By analysing the computation of $\Psi_0 \cdot F_0^+ \cdot G_0 = c$, initially just at the 'top level' (that is, without delving into the computations of the type 1 arguments passed to F_0^+ and G_0), we are able to glean some 'neighbourhood information' about G_0 which helps to secure aspects of this computation (and will also secure the corresponding computation for G_1 once the latter has been constructed). In the course of this, we will also have replaced F_0^+ by the next iteration F_1^+.

However, the information about G_0 gathered so far does not by itself suffice to secure the entire computation: the top-level computation will typically rely on certain information about the arguments passed to F and G; and since these may themselves involve calls to G, some further constraints on G_0 may be needed to secure this information. We are thus led to repeat our analysis for certain subcomputations associated with the arguments to F and G—and so on recursively to whatever depth is required. This is done in Section 3.2.

A key step in the proof is to observe that since Ψ_0 is LWF, this entire construction will eventually bottom out in a situation where no further subcomputations need to be analysed (there is a crucial appeal to König's lemma here). We record what happens at this final stage of the construction in Section 3.3.

At the end of this computation analysis, we are left with two things. First, in the course of the analysis, the value of F we are considering will have been successively refined via an approximation process, and at the end we are able to fix on the definitive value (denoted by F_∞) which we shall use to obtain a contradiction. Second, our analysis as a whole generates enough 'neighbourhood conditions' on G_0 to secure the entire computation: that is, we obtain a certain neighbourhood $\mathcal{G} \subseteq \mathsf{SP}^0(2)$ containing G_0 such that for any $G \in \mathcal{G}$ we have $\Psi_0 \cdot F_\infty \cdot G = c$. The definition of $\mathcal{G}$ together with this key property are established in Section 3.4.

The remainder of the proof proceeds along the lines already indicated. In Section 3.5, we draw on the above analysis to construct a certain procedure G_1 designed to force Φ_0 to explore parts of $\mathcal{T}(F_\infty)$ beyond the reach of Ψ_0. In Section 3.6 we verify the required properties of G_1, namely that $G_1 \in \mathcal{G}$ (so that $\Psi_0 \cdot F_\infty \cdot G_1 = c$) and also that $\Phi_0 \cdot F_\infty \cdot G_1$ yields some value K different from c. Since this latter fact will hold for *any* genuine bar recursor Φ, this establishes that Ψ is not a genuine bar recursor.

We now proceed to the formal details of the proof.

3.1 Computation analysis: the top level

As indicated above, we begin by supposing that $\Phi, \Psi \in \mathsf{SP}^0(2 \to 2 \to 1)$ are simplified weak Kohlenbach bar recursors in the sense of Definition 21, and assuming for contradiction that Ψ is LWF. We set $\Phi_0 = \lambda FG.\, \Phi FG\langle\rangle \in \mathsf{SP}^0(2 \to 2 \to 0)$ (or more formally $\Phi_0 = \lambda FG.\, \Phi \cdot F^\eta \cdot G^\eta \cdot \langle\rangle$), and similarly $\Psi_0 = \lambda FG.\, \Psi FG\langle\rangle$. Clearly Ψ_0 is LWF by Theorem 9(i).

In general, if t is any NSP term possibly containing F^2, G^2 free, and $F', G' \in \mathsf{SP}^0(2)$, we shall write $t[F', G']$ for the closed term obtained from t by instantiating F, G to F', G' and then evaluating. (For instance, if t is a procedure then formally $t[F', G'] = (\lambda FG.t) \cdot F' \cdot G'$.)

To start our construction, we consider the T_0^{str}-definable procedures

$$\begin{aligned} G_0 &= \lambda g.\, \mathtt{case}\ g(0)\ \mathtt{of}\ (i \Rightarrow 2i)\ , \\ F_0^+ &= \lambda f.\, \mathtt{case}\ f(0)\ \mathtt{of}\ (i \Rightarrow \langle i \rangle)\ . \end{aligned}$$

The purpose of the doubling in the definition of G_0 is hard to motivate here, but will emerge in Section 3.6. The functional F_0^+ represents a very simple well-founded tree: note that $\langle\rangle$ is not a leaf in $\mathcal{T}(F_0^+)$, but $\langle x_0 \rangle$ is a leaf for every $x_0 \in \mathbb{N}$. Note too that the label on such a leaf node tells us the position of the node in the tree: this will be useful later for ensuring that certain numbers arising from different parts of the tree are always distinct.

The definition of simplified weak bar recursor now implies that $\Psi_0 \cdot F_0^+ \cdot G_0$ now evaluates to a certain $c \in \mathbb{N}$, or more formally to $\lambda.c$. (In fact $c = 4\langle 0 \rangle + 2$, but we will not need this information.) By continuity of application, we may pick $k^0 > 0$ large enough that $\Psi_0 \cdot F_0 \cdot G_0 = c$, where

$$F_0 \ = \ \lambda f.\, \mathtt{case}\ f(0)\ \mathtt{of}\ (i < k^0 \Rightarrow \langle i \rangle \mid i \geq k^0 \Rightarrow \bot)$$

(extending our notation for `case` expressions in an obvious way). Note that $F_0 \sqsubseteq F_0^+$. We shall actually use F_0 (rather than F_0^+) as the first step in our approximative construction of a suitable F.

Let us now look at the computation of $\Psi_0 \cdot F_0 \cdot G_0 = c$. This will take the form of a head reduction of $\Psi_0 F_0 G_0$, and by inspection of the reduction rules in Section 2.2, it is clear that this will follow a path through the syntax tree of Ψ_0 consisting of a finite sequence of calls to F or G (in any order), and leading to a leaf c. For example, such a path might have the form

$$\lambda FG.\, \mathtt{case}\ F(f_0^0)\ \mathtt{of}\ u_0^0 \Rightarrow \mathtt{case}\ G(g_0^0)\ \mathtt{of}\ v_0^0 \Rightarrow \mathtt{case}\ F(f_1^0)\ \mathtt{of}\ u_1^0 \Rightarrow \cdots \Rightarrow c\ .$$

where the f_i^0 and g_i^0 are themselves type 1 procedures which appear syntactically within Ψ_0 and which may contain F, G as free variables (the superscript indicates that we are here analysing the computation at 'level 0'). We can view the tracing of such a path through Ψ as the 'top level computation'; in addition to this, there will be subcomputations showing (for instance) that $(Ff_0^0)[F_0, G_0]$ evaluates to u_0^0 and $(Gg_0^0)[F_0, G_0]$ evaluates to v_0^0,

Let $f_0^0, \ldots, f_{l^0-1}^0$ be the complete list of such procedures appearing as arguments to F along this computation path, with $u_0^0, \ldots, u_{l^0-1}^0$ the corresponding outcomes when F, G are instantiated to F_0, G_0. Likewise, let $g_0^0, \ldots, g_{n^0-1}^0$ be the list of procedures appearing as arguments to G on this path, with $v_0^0, \ldots, v_{n^0-1}^0$ the corresponding outcomes.

Of course, when $F = F_0$ and $G = G_0$, the procedures f_i^0 and g_i^0 will be interrogated only on the argument 0. This suggests that in order to 'secure' the whole computation, we will also want to analyse the computations of $(f_i^0 0)[F_0, G_0] = u_i^0$ and $(g_i^0 0)[F_0, G_0] = v_i^0$ for each i. In fact, we shall do more: in order to give ourselves sufficient room for

manoeuvre to construct the contrary example G_1 below (with the assurance of a similar evaluation behaviour for G_1), we shall analyse the behaviour of each g_i^0 on all integer arguments z up to a certain *modulus* m^0. In fact, it will suffice to take

$$m^0 > k^0 + n^0 + 1 \ .$$

Again, this condition is hard to motivate at this stage; the reason for it will emerge during the construction of G_1 in Section 3.5, at the point where we select x_0, the first step in our critical path through $\mathcal{T}(F_\infty)$.

In order to proceed further, we need to extend our approximation to F. First, extend the procedure F_0 to a T_0^{str}-definable F_1^+:

$$\begin{array}{rcl} F_1^+ & = & \lambda f.\ \ \mathtt{case}\ f(0)\ \mathtt{of}\ (i_0 < k^0 \Rightarrow \langle i_0 \rangle \mid i_0 \geq k^0 \Rightarrow \\ & & \qquad \mathtt{case}\ f(1)\ \mathtt{of}\ (i_1 \Rightarrow \langle i_0, i_1 \rangle)) \ . \end{array}$$

(It is an easy exercise to verify that this is indeed T_0^{str}-definable.) The idea is that $\langle x_0 \rangle$ will be a leaf node in $\mathcal{T}(F_1^+)$ when $x_0 < k^0$, but elsewhere $\mathcal{T}(F_1^+)$ will have depth 2.

We now consider the computation of $\Psi_0 \cdot F_1^+ \cdot G_0$. Since $F_1^+ \sqsupseteq F_0$, this follows the same path through Ψ_0 as before and features syntactically the same type 1 procedures f_i^0 and g_i^0 and the same outcomes u_i^0, v_i^0. Furthermore:

Lemma 25 *(i) For any $i < n^0$ and any $z \in \mathbb{N}$, the evaluation of $g_i^0[F_1^+, G_0] \cdot z$ yields a natural number, which we denote by r_{iz}^0.*

(ii) For any $i < l^0$ and any $z \in \mathbb{N}$, the evaluation of $f_i^0[F_1^+, G_0] \cdot z$ yields a natural number, which we denote by q_{iz}^0.

Proof: (i) Suppose for contradiction that $g_i^0[F_1^+, G_0](z) = \bot$ for some i, z, and let $G_0' = \lambda g.\ \mathtt{case}\ g(z)\ \mathtt{of}\ (j \Rightarrow G_0(g))$. Clearly G_0' is T_0^{str}-definable. Also $G_0' \preceq G_0$ in the extensional preorder on NSPs, so by Theorem 6 we have

$$(G_0(g_i^0))[F_1^+, G_0'] \ \sqsubseteq\ G_0'(g_i^0[F_1^+, G_0]) \ = \ \bot \ .$$

Moreover, for each application $F(f_j^0)$ (respectively $G(g_j^0)$) occurring before $G(g_i^0)$ in the path in question, we have $(F(f_j^0))[F_1^+, G_0'] \sqsubseteq u_j^0$ (respectively $(G(g_j^0))[F_1^+, G_0'] \sqsubseteq v_j^0$), whence it is clear that $\Psi_0 \cdot F_1^+ \cdot G_0'$ is undefined. But this contradicts Proposition 22, since both F_1^+, G_0' are T_0^{str}-definable, $\mathcal{T}(F_1^+)$ is well-founded and $\langle\,\rangle \in \mathcal{T}(F_1^+)$.

The proof of (ii) is precisely similar. □

We shall make use of part (i) of the above lemma for all $z < m^0$, and of part (ii) only when $z = 0$. (The apparently superfluous use of z in the latter case is intended to mesh with a more general situation treated below.)

We may now make explicit some significant properties of G_0 in the form of *neighbourhoods*, using the information gleaned so far. For each $i < n^0$, define

$$\begin{array}{rcl} V_i^0 & = & \{g \in \mathsf{SP}^0(1) \ \mid\ \forall z < m^0.\, g \cdot z = r_{iz}^0\} \ , \\ \mathcal{G}_i^0 & = & \{G \in \mathsf{SP}^0(2) \ \mid\ \forall g \in V_i^0.\, G \cdot g = v_i^0\} \ . \end{array}$$

Clearly $G_0 \in \mathcal{G}_i^0$ for each i, because G_0 interrogates its argument only at 0. Also $g_i^0[F_1^+, G_0] \in V_i^0$ for each $i < n^0$. This completes our analysis of the computation at top level; we shall refer to this as the *depth* 0 *analysis*.

The idea is that the sets $\mathcal{G}_i^0$ will form part of a system of neighbourhoods recording all the necessary information about G_0; we will then be free to select any G_1 from the intersection of these neighbourhoods knowing that the computation will proceed as before. As things stand, the neigbourhoods $\mathcal{G}_i^0$ do not achieve this: for an arbitrary G in all these neigbourhoods, there is no guarantee that the value of each $g_i^0(z)$ at F_1^+ and G will agree with its value at F_1^+ and G_0 (and similarly for each $f_i^0(0)$). To secure the whole computation, we therefore need a deeper analysis of these subcomputations in order to nail down the precise properties of G_0 on which these rely.

3.2 Computation analysis: the step case

The idea is that we repeat our analysis for each of the finitely many computations of

$$f_i^0(z)[F_1^+, G_0] \;\; (i < l^0,\ z < 1)\,, \qquad g_i^0(z)[F_1^+, G_0] \;\; (i < n^0,\ z < m^0)\,.$$

The analysis at this stage is in fact illustrative of the general analysis at depth w, assuming we have completed the analysis at depth $w-1$. For notational simplicity, however, we shall concentrate here on the depth 1 analysis, adding a few brief remarks on the depth 2 analysis in order to clarify how the construction works in general.

First, since each of the above computations yields a numeral q_{iz}^0 or r_{iz}^0 as appropriate, we may choose k^1 such that all these computations yield the same results when F_1^+ is replaced by

$$\begin{aligned} F_1 \;\; = \;\; \lambda f.\;\; & \texttt{case}\ f(0)\ \texttt{of}\ (i_0 < k^0 \Rightarrow \langle i_0 \rangle \mid i_0 \geq k^0 \Rightarrow \\ & \texttt{case}\ f(1)\ \texttt{of}\ (i_1 < k^1 \Rightarrow \langle i_0, i_1 \rangle \mid i_1 \geq k^1 \Rightarrow \bot))\,. \end{aligned}$$

Note in passing that F_0 no longer suffices here: there will be computations of values for $g_i^0(z)$ that did not feature anywhere in the original computation of $\Psi_0 \cdot F_0 \cdot G_0$.

Everything we have said about the main computation and its subcomputations clearly goes through with F_1^+ replaced by F_1. So let us consider the shape of the computations of

$$f_i^0(z)[F_1, G_0] \;\; (i < l^0,\ z < 1)\,, \qquad g_i^0(z)[F_1, G_0] \;\; (i < n^0,\ z < m^0)\,.$$

At top level, each of these consists of a finite sequence of applications of F_1 and G_0 (in any order), leading to the result q_{iz}^0 or r_{iz}^0. Taking all these computations together, let $f_0^1, \ldots, f_{l^1-1}^1$ and $g_0^1, \ldots, g_{n^1-1}^1$ respectively denote the (occurrences of) type 1 procedures to which F_1 and G_0 are applied, with $u_0^1, \ldots, u_{l^1-1}^1$ and $v_0^1, \ldots, v_{n^1-1}^1$ the corresponding outcomes. Although we will not explicitly track the fact in our notation, we should consider each of the f_j^1 and g_j^1 as a 'child' of the procedure f_i^0 or g_i^0 from which it arose. Note that if g_j^1 is a child of f_i^0 (for example), then just as Ff_i^0 appears as a subterm within the syntax tree of Ψ_0, so Gg_j^1 appears as a subterm within the syntax tree of f_i^0. Thus, each of the f_j^1 and g_j^1 corresponds to a path in Ψ_0 with at least two left branches.

We now select a suitable modulus for our analysis of the g_i^1. Choose

$$m^1 > k^1 + n_0 + n_1 + 2\,, \quad m^1 \geq m^0\,.$$

(Again, the reason for this choice will emerge in Section 3.5.) Extend F_1 to the T_0^{str}-definable functional

$$\begin{array}{rcl} F_2^+ & = & \lambda f.\ \ \mathtt{case}\ f(0)\ \mathtt{of}\ (i_0 < k^0 \Rightarrow \langle i_0\rangle \mid i_0 \geq k^0 \Rightarrow \\ & & \quad \mathtt{case}\ f(1)\ \mathtt{of}\ (i_1 < k^1 \Rightarrow \langle i_0, i_1\rangle \mid i_1 \geq k^1 \Rightarrow \\ & & \quad \mathtt{case}\ f(2)\ \mathtt{of}\ (i_2 \Rightarrow \langle i_0, i_1, i_2\rangle\ )))\,. \end{array}$$

Replacing F_1 by F_2^+ preserves all the structure established so far, and just as in Lemma 25 we have that $g_i^1(z)$ at F_2^+, G_0 yields a numeral r_{iz}^1 for each $i < n^1$ and $z < m^1$; similarly $f_i^1(z)$ at F_2^+, G_0 yields a numeral q_{iz}^1 for each $i < l^1$ and $z < 2$. (In fact, it is superfluous to consider $f_i^1(1)$ in cases where $f_i^1(0) < k^0$, but it simplifies notation to use 2 here as our uniform modulus of inspection for the f_i^1.) We may now augment our collection of neighbourhoods by defining

$$\begin{array}{rcl} V_i^1 & = & \{g \in \mathsf{SP}^0(1) \ \mid\ \forall z < m^1.\, g \cdot z = r_{iz}^1\}\,, \\ \mathcal{G}_i^1 & = & \{G \in \mathsf{SP}^0(2) \ \mid\ \forall g \in V_i^1.\, G \cdot g = v_i^1\}\,. \end{array}$$

for each $i < n^1$; note once again that $G_0 \in \mathcal{G}_i^1$ and that $g_1^0[F_2^+, G_0] \in V_i^1$ for each i. This completes our analysis of the computation at depth 1.

At the next stage, we choose k^2 so that the above all holds with F_2^+ replaced by

$$\begin{array}{rcl} F_2 & = & \lambda f.\ \ \mathtt{case}\ f(0)\ \mathtt{of}\ (i_0 < k^0 \Rightarrow \langle i_0\rangle \mid i_0 \geq k^0 \Rightarrow \\ & & \quad \mathtt{case}\ f(1)\ \mathtt{of}\ (i_1 < k^1 \Rightarrow \langle i_0, i_1\rangle \mid i_1 \geq k^1 \Rightarrow \\ & & \quad \mathtt{case}\ f(2)\ \mathtt{of}\ (i_2 < k^2 \Rightarrow \langle i_0, i_1, i_2\rangle \mid i_2 \geq k^2 \Rightarrow \bot)))\,. \end{array}$$

We now repeat our analysis for each of the computations of

$$f_i^1(z)[F_2, G_0]\ \ (i < l^1,\ z < 2)\,, \qquad g_i^1(z)[F_2, G_0]\ \ (i < n^1,\ z < m^1)\,.$$

Having identified the relevant type 1 procedures $f_0^2, \ldots, f_{l^2-1}^2$ and $g_0^2, \ldots, g_{n^2-1}^2$ that feature as arguments to F and G, we pick

$$m^2 > k^2 + n^0 + n^1 + n^2 + 3\,, \quad m^2 \geq m^1\,,$$

and use this to define suitable sets $V_i^2, \mathcal{G}_i^2$ for $i < n^2$. By this point, it should be clear how our construction may be continued to arbitrary depth.

3.3 Computation analysis: the bottom level

The crucial observation is that this entire construction eventually bottoms out. Indeed, using h as a symbol that can ambivalently mean either f or g (and likewise H for F or G), we have that for any sequence $h_{i^0}^0, h_{i^1}^1, \ldots$ of type 1 procedures where each $h_{i^{w+1}}^{w+1}$ is a child of $h_{i^w}^w$, the syntax tree of Ψ_0 contains the descending sequence of subterms

$H^0 h^0_{i^0}, H^1 h^1_{i^1}, \ldots$. Since Ψ_0 is LWF by assumption, any such sequence must eventually terminate. Moreover, the tree of all such procedures h^w_i is finitely branching, so by König's lemma it is finite altogether.

Let us see explicitly what happens at the last stage of the construction. For some depth d, we will have constructed the $f^d_i, g^d_i, u^d_i, v^d_i$ as usual, along with m^d, F^+_{d+1}, the numbers r^d_{iz}, q^d_{iz} and the neighbourhoods $\mathcal{G}^d_i$, but will then discover that $l^{d+1} = n^{d+1} = 0$: that is, none of the relevant computations of $f^d_i(z)$ or $g^d_i(z)$ (relative to F^+_{d+1} and G_0) themselves perform calls to F or G.

At this point, we may settle on F^+_{d+1} as the definitive version of F to be used in our counterexample, and henceforth call it F_∞. Explicitly:

$$\begin{aligned} F_\infty \;=\; \lambda f.\;\; & \texttt{case } f(0) \texttt{ of } (i_0 < k^0 \Rightarrow \langle i_0 \rangle \mid i_0 \geq k^0 \Rightarrow \\ & \texttt{case } f(1) \texttt{ of } (i_1 < k^1 \Rightarrow \langle i_0, i_1 \rangle \mid i_1 \geq k^1 \Rightarrow \\ & \ldots \\ & \texttt{case } f(d) \texttt{ of } (i_d < k^d \Rightarrow \langle i_0, \cdots, i_d \rangle \mid i_d \geq k^d \Rightarrow \\ & \texttt{case } f(d+1) \texttt{ of } (i_{d+1} \Rightarrow \langle i_0, \cdots, i_{d+1} \rangle\,))\cdots)) \;. \end{aligned}$$

Clearly F_∞ is T^{str}_0-definable and $F_\infty \sqsupseteq F_w$ for $w \leq d$. It is also clear from the above definition how F_∞ represents a certain well-founded tree $\mathcal{T}(F_\infty)$ of depth $d{+}2$. Note that if $f(0) \geq k^0, \ldots, f(d) \geq k^d$ then $F_\infty \cdot f = \langle f(0), \ldots, f(d{+}1) \rangle$; indeed $\langle f(0), \ldots, f(d{+}1) \rangle$ is a leaf in $\mathcal{T}(F)$. It is this portion of the tree, not visited by any of the computations described so far, that we shall exploit when we construct our counterexample G_1.

3.4 The critical neighbourhood of G_0

We may now define the *critical neighbourhood* $\mathcal{G} \subseteq \mathsf{SP}^0(2)$ by

$$\mathcal{G} \;=\; \bigcap_{w \leq d,\; i < n^w} \mathcal{G}^w_i \;.$$

Clearly $G_0 \in \mathcal{G}$ by construction. Moreover, the following lemma shows that $\mathcal{G}$ provides enough constraints to secure the result of the entire computation:

Lemma 26 *For all $G \in \mathcal{G}$ and all $w \leq d$, we have:*

1. $f^w_i(z)[F_\infty, G] = q^w_{iz}$ *for all* $i < l^w$ *and* $z \leq w$.
2. $g^w_i(z)[F_\infty, G] = r^w_{iz}$ *for all* $i < n^w$ *and* $z < m^w$.
3. $F(f^w_i)[F_\infty, G] = u^w_i$ *for all* $i < l^w$.
4. $G(g^w_i)[F_\infty, G] = v^w_i$ *for all* $i < n^w$.
5. $\Psi_0 \cdot F_\infty \cdot G = c$.

Proof: We prove claims 1–4 simultaneously by downwards induction on w. For $w = d$, claims 1 and 2 hold because the computations in question make no use of F_∞ or G. For any w, claim 1 implies claim 3: F_w was chosen so that (among other things)

$F_w(f_i^w[F_w, G_0]) = u_i^w$ is defined; moreover, F_w interrogates its argument only on $0, \ldots, w$ at most, so the established values of $f_i^w(z)[F_\infty, G]$ for $z \leq w$ suffice to ensure that $F_w(f_i^w[F_\infty, G]) = u_i^w$, and hence that $F_\infty(f_i^w[F_\infty, G]) = u_i^w$. Likewise, claim 2 implies claim 4, since $G \in \mathcal{G}_i^w$ by hypothesis, and the established values of g_i^w secure that $g_i^w \in V_i^w$ (at F_∞ and G).

Assuming claims 3 and 4 hold for $w+1$, it is easy to see that claims 1 and 2 hold for w: the relevant top-level computation may be reconstituted from left to right leading to the result q_{iz}^w or r_{iz}^w. Applying the same argument one last time also yields claim 5. □

3.5 The counterexample G_1

It remains to construct our contrary example $G_1 \in \mathcal{G}$. The idea is that G_1 will be chosen so that according to the definition of simplified bar recursion, some value $K \neq c$ will be generated at depth $d+1$ of the tree $\mathcal{T}(F_\infty)$ and then propagated up to the surface of the computation via nested applications of G_1. We work with paths beyond the horizon defined by $k^0, k^1, \ldots$ to ensure that we do not encounter a leaf prematurely, and also exploit the choice of moduli m^w to ensure that the type 1 functions at intermediate levels steer clear of the sets V_i^w.

Recall that Φ is assumed to be a genuine simplified bar recursor within SP^0. Set $\phi_0 = \Phi \cdot F_\infty \cdot G_0 \in \mathsf{SP}^0(1)$. If x is any sequence code $\langle x_0, \ldots, x_{n-1}\rangle$ and $z \in \mathbb{N}$, we shall write $x.z$ for the sequence code $\langle x_0, \ldots, x_{n-1}, z\rangle$, so that $x.z$ is the number computed by $\mathrm{add} \cdot x \cdot z$.

Since $G_0 = \lambda g.2g(0)$ and the leaf function has been fixed at $x \mapsto 2x+1$, we have that for any sequence code x, $\phi_0 \cdot x$ will take one of the values

$$2x+1\ ,\quad 2(2(x.0)+1)\ ,\quad 4(2(x.0.0)+1)\ ,\quad 8(2(x.0.0.0)+1)\ ,\quad \ldots\ ,$$

according to where a leaf of $\mathcal{T}(F_\infty)$ appears in the sequence x, $x.0$, $x.0.0$, ... In particular, for any fixed j, if we know that $j < |x|$, we can recover x_j from $\phi_0 \cdot x$ and even from $\theta(\phi_0 \cdot x)$, where $\theta(n)$ denotes the unique odd number such that $n = 2^t.\theta(n)$ for some t. We shall write $x.0^t$ for the result of appending t occurrences of 0 to the sequence number x; note that $\theta(\phi_0 \cdot x)$ will have the value $2(x.0^t)+1$ for some t.

We construct a finite path $x_0, x_1, \ldots, x_d$ through the tree for F_∞ in the following way, along with associated numbers $y_0, y_1, \ldots, y_d, y_{d+1}$. Start by setting $y_0 = \phi_0 \cdot \langle 0\rangle$. Next, note that the mappings $z \mapsto \phi_0 \cdot \langle z, 0\rangle$ and $z \mapsto \theta(\phi_0 \cdot \langle z, 0\rangle)$ are injective; so because $m^0 > k^0 + n^0 + 1$, we may pick x_0 with $k^0 \leq x_0 < m^0$ such that:

- $y_1 = \phi_0 \cdot \langle x_0, 0\rangle$ differs from $g_i^0(0)$ (more precisely from r_{i0}^0) for each $i < n^0$,
- $\theta(y_1) = \theta(\phi_0 \cdot \langle x_0, 0\rangle)$ differs from $\theta(y_0)$.

Likewise, the mapping $z \mapsto \theta(\phi_0 \cdot \langle x_0, z, 0\rangle)$ is injective, so since $m^1 > k^1 + n^0 + n^1 + 2$ we may pick x_1 with $k^1 \leq x_1 < m^1$ such that

- $y_2 = \phi_0 \cdot \langle x_0, x_1, 0\rangle$ is different from all r_{i0}^0 and $r_{i'0}^1$ where $i < n^0$, $i' < n^1$,
- $\theta(y_2)$ is different from $\theta(y_0)$ and $\theta(y_1)$.

In general, we pick x_w with $k^w \leq x_w < m^w$ such that

- $y_{w+1} = \phi_0 \cdot \langle x_0, \ldots, x_w, 0\rangle$ is different from all r^u_{i0} with $u \leq w$ and $i < n^u$,
- $\theta(y_{w+1})$ is different from $\theta(y_0), \ldots, \theta(y_w)$.

In each case, the first condition ensures that the type 1 function $\Lambda z.\, \phi_0 \cdot \langle \vec{x}, z\rangle$ steers clear of the sets V^u_i, so that the functional G_1 to be defined below remains within $\mathcal{G}$. The second condition will ensure that the nested calls to G_1 do not interfere with one another in their role of propagating the special value K. Since $x_w \geq k^w$ for each $w \leq d$, we have that $\langle x_0, \ldots, x_d, 0\rangle$ is a leaf of $\mathcal{T}(F_\infty)$.

We now take K to be some natural number larger than any that has featured in the construction so far, and in particular different from c, and define

$$\begin{array}{rcll} G_1 & = & \lambda g. & \texttt{case}\ g(0)\ \texttt{of}\ (\\ & & & y_{d+1} \Rightarrow K \\ & & \mid & y_d \Rightarrow \texttt{case}\ g(x_d)\ \texttt{of}\ (K \Rightarrow K \mid j \Rightarrow 2i) \\ & & \mid & \cdots \\ & & \mid & y_1 \Rightarrow \texttt{case}\ g(x_1)\ \texttt{of}\ (K \Rightarrow K \mid j \Rightarrow 2i) \\ & & \mid & y_0 \Rightarrow \texttt{case}\ g(x_0)\ \texttt{of}\ (K \Rightarrow K \mid j \Rightarrow 2i) \\ & & \mid & i \Rightarrow 2i \\ & &) & . \end{array}$$

Here we understand i, j as 'pattern variables' that catch all cases not handled by the preceding clauses. In particular, the clauses $j \Rightarrow 2i$, $i \Rightarrow 2i$ mean that unless g possesses some special property explicitly handled by some other clause, we will have $G_1 \cdot g = 2(g \cdot 0) = G_0 \cdot g$. It is straightforward to verify that G_1 is T^{str}_0-definable, bearing in mind the availability of *ifzero* in T^{str}_0 (see the discussion in Section 2.1.1).

3.6 Properties of G_1

We first check that G_1 falls within the critical neighbourhood:

Lemma 27 *$G_1 \in \mathcal{G}$, whence $\Psi \cdot F_\infty \cdot G_1 \cdot \langle\,\rangle = c$.*

Proof: Suppose $w \leq d$ and $i < n^w$; we will show $G_1 \in \mathcal{G}^w_i$. Consider an arbitrary $g \in V^w_i$ (note that g is not assumed to represent a total function); we want to show that $G_1 \cdot g = v^w_i$. From the definition of V^w_i we have $g \cdot 0 = r^w_{i0}$, so for $u > w$, we have $g \cdot 0 \neq y_u$ by choice of y_u. If also $g \cdot 0 \neq y_u$ for each $u \leq w$ then $G_1(g) = 2(g \cdot 0) = G_0(g) = v^w_i$ as required. If $g \cdot 0 = y_u$ for some $u \leq w$, then $G_1(g) = \texttt{case}\ g(x_u)\ \texttt{of}\ (K \Rightarrow K \mid i \Rightarrow 2(g \cdot 0))$. However, since $x_u < m^u \leq m^w$ we have $g(x_u) = r^w_{ix_u}$, and K was assumed to be larger than this, so once again $G_1(g) = 2(g \cdot 0) = v^w_i$.

By claim 5 of Lemma 26, it follows that $\Psi \cdot F_\infty \cdot G_1 \cdot \langle\,\rangle = c$. □

We now work towards showing that, by contrast, $\Phi \cdot F_\infty \cdot G_1 \cdot \langle\rangle = K$. Set $\phi_1 = \Phi \cdot F_\infty \cdot G_1$, and for $w \leq d+1$, denote $\langle x_0, \ldots, x_{w-1}\rangle$ by x^w.

Lemma 28 *$\phi_1 \cdot (x^w.0) = y_w$ for all $w \leq d+1$.*

PROOF: Recall that $y_w = \phi_0 \cdot (x^w.0)$, which has the form $2^t.s$ where $s = \theta(y_w) = \phi_0 \cdot (x^w.0.0^t)$ and $x^w.0.0^t$ is a leaf for F_∞, so that $s = 2(x^w.0.0^t) + 1$. Note also that if $0 < t' \leq t$ then $\phi_0 \cdot (x^w.0.0^{t'}) = 2^{t-t'}.s$, which is distinct from y_w (this is the point of the doubling in the definition of G_0), and also from all the other y_u since $\theta(y_0), \ldots, \theta(y_{d+1})$ are all distinct.

We may now see by reverse induction on $t' \leq t$ that $\phi_1 \cdot (x^w.0.0^{t'}) = 2^{t-t'}.s$. When $t' = t$, this holds because $x^w.0.0^t$ is a leaf for F_∞ so $\phi_1 \cdot (x^w.0.0^t) = \phi_0 \cdot (x^w.0.0^t)$. Assuming this holds for $t'+1$ with $0 \leq t' < t$, because $x^w.0.0^{t'}$ is not a leaf we have

$$\begin{aligned}
\phi_1 \cdot (x^w.0.0^{t'}) &= G_1 \cdot (\lambda z.\phi_1 \cdot (x^w.0.0^{t'}.z)) \\
&= \texttt{case}\ \phi_1 \cdot (x^w.0.0^{t'}.0)\ \texttt{of}\ (\cdots \mid i \Rightarrow 2i) \\
&= \texttt{case}\ 2^{t-(t'+1)}.s\ \texttt{of}\ (\cdots \mid i \Rightarrow 2i) \\
&= 2^{t-t'}.s\ ,
\end{aligned}$$

using the observation that $2^{t-(t'+1)}.s$ is distinct from all of the y_u.

In particular, $\phi_1 \cdot (x^w.0) = 2^t.s = y_w$, so the lemma is established. □

Lemma 29 *$\phi_1 \cdot x^w = K$ for all $0 \leq w \leq d+1$.*

PROOF: By reverse induction on w. For the case $w = d+1$, we have by the previous lemma that $\phi_1 \cdot (x^{d+1}.0) = y_{d+1}$, and since x^{d+1} is not a leaf for F_∞, we have

$$\begin{aligned}
\phi_1 \cdot x^{d+1} &= G_1(\lambda z.\, \phi_1 \cdot (x^{d+1}.z)) \\
&= \texttt{case}\ \phi_1 \cdot (x^{d+1}.0)\ \texttt{of}\ (y_{d+1} \Rightarrow K \mid \cdots) \\
&= K\ .
\end{aligned}$$

For $w < d+1$, again we have by the previous lemma that $\phi_1 \cdot (x^w.0) = y_w$, and the induction hypothesis gives us $\phi_1 \cdot (x^w.x_w) = \phi_1 \cdot (x^{w+1}) = K$. Since x^w is not a leaf for F_∞, we have

$$\begin{aligned}
\phi_1 \cdot x^w &= G_1(\lambda z.\, \phi_1 \cdot (x^w.z)) \\
&= \texttt{case}\ \phi_1 \cdot (x^w.0)\ \texttt{of} \\
&\qquad (\cdots \mid y_w \Rightarrow \texttt{case}\ \phi_1 \cdot (x^w.x_w)\ \texttt{of}\ (K \Rightarrow K \mid \cdots) \mid \cdots) \\
&= K\ . \qquad \square
\end{aligned}$$

In particular, when $w = 0$ we have $\phi_1 \cdot \langle\,\rangle = \phi_1 \cdot x^0 = K$. Combining this with Lemma 27, we have

$$\Psi \cdot F_\infty \cdot G_1 \cdot \langle\rangle \;=\; c \;\neq\; K \;=\; \phi_1 \cdot \langle\,\rangle \;=\; \Phi \cdot F_\infty \cdot G_1 \cdot \langle\rangle\ .$$

Since this argument applies for any genuine weak simplified bar recursor Ψ, we may conclude that Ψ is not a restricted bar recursor after all. This completes the proof of Theorem 24.

4 Other models

Finally, we show how our non-definability result now transfers readily to settings other than SP^0, both partial and total. The combined message of these results will be that bar recursion is not computable in $\mathrm{T} + \mathit{min}$ or W in any reasonable sense whatever, however one chooses to make such a statement precise.

4.1 Partial models

It is relatively easy to transfer Theorem 24 to other 'partial' settings, by which we here mean simply-typed λ-algebras $\mathbf{A}$ with $\mathbf{A}(\mathtt{N}) \cong \mathbb{N}_\perp$. As a first step, it is convenient to detach our main theorem from SP^0 and present its content in purely syntactic terms.

Let $F : 2$ be a closed term of T_0^{str}. Using the T_0^{str} program 'basic' introduced in Section 2.3, and reinstating the hat notation for programming language numerals, we may say a sequence $\vec{x} \in \mathbb{N}^*$ satisfies the Kohlenbach bar condition w.r.t. F if the closed T_0^{str} terms

$$F(\mathrm{basic}(\widehat{\langle \vec{x} \rangle}, \widehat{0})) \ , \qquad F(\mathrm{basic}(\widehat{\langle \vec{x} \rangle}, \widehat{1}))$$

evaluate to the same numeral; we may thus define $\mathcal{T}^K(F)$ to be the tree of sequences $\vec{x}$ such that no proper prefix of $\vec{x}$ satisfies this bar condition. It is clear that this purely syntactic definition of $\mathcal{T}^K(F)$ agrees with Definition 16 for NSPs: if $[\![F]\!]$ is the denotation of F in SP^0, then $\mathcal{T}^K(F) = \mathcal{T}^K([\![F]\!])$ by the adequacy of $[\![-]\!]$.

This allows us to reformulate the content of Theorem 24 syntactically as follows. Here we write $=$ to mean that the closed programs on either side evaluate to the same numeral.

Theorem 30 *There is no closed* $\mathrm{T} + \mathit{min}$ *term* $\mathsf{BR} : 2 \to 2 \to 1$ *such that the following hold for all closed* T_0^{str} *terms* $F, G : 2$ *with* $\mathcal{T}^K(F)$ *well-founded, and for all* $\vec{x} \in \mathcal{T}^K(F)$*:*

$$\begin{array}{lcl} \mathsf{BR}\ F\ G\ \widehat{\langle \vec{x} \rangle} & = & \widehat{2\langle \vec{x} \rangle + 1} \quad \textit{if } \vec{x} \in \mathcal{T}^K(F)^l \ , \\ \mathsf{BR}\ F\ G\ \widehat{\langle \vec{x} \rangle} & = & G(\lambda z^{\mathtt{N}}.\ \mathsf{BR}\ F\ G\ (\mathrm{add}\ \widehat{\langle \vec{x} \rangle}\ z)) \quad \textit{if } \vec{x} \in \mathcal{T}^K(F)^n \ . \end{array}$$

Proof: If such a term BR existed, then by adequacy of $[\![-]\!]$, $\Phi = [\![\mathsf{BR}]\!] \in \mathsf{SP}^0$ would be a $\mathrm{T} + \mathit{min}$ definable simplified weak Kohlenbach recursor, contradicting Theorem 24. $\square$

Now suppose $\mathbf{A}$ is any simply typed λ-algebra equipped with elements $0, 1, \ldots$, *suc*, *pre*, *ifzero*, rec_σ, *min* of the appropriate types, such that the induced interpretation $[\![-]\!]_{\mathbf{A}}$ of $\mathrm{T} + \mathit{min}$ in $\mathbf{A}$ is *adequate*: that is, for closed programs $M : \mathtt{N}$ and $n \in \mathbb{N}$, we have $[\![M]\!]_{\mathbf{A}} = n \in \mathbf{A}(\mathtt{N})$ iff $M \rightsquigarrow^* n$. Note that this requires $\mathbf{A}(\mathtt{N})$ to contain elements other than the numerals, since in the presence of *min*, diverging programs are possible. In most cases of interest, we will have $\mathbf{A}(\mathtt{N}) \cong \mathbb{N}_\perp$: typical examples include the Scott model PC of partial continuous functionals, its effective submodel $\mathsf{PC}^{\mathrm{eff}}$, the model SF of PCF-sequential functionals (arising as the extensional quotient of SP^0) and its effective submodel $\mathsf{SF}^{\mathrm{eff}}$ of PCF-computable functionals.

In this setting, we have a notion of T_0^{str}-definable element of $\mathbf{A}$, so Definitions 16, 18 and 21 immediately relativize to $\mathbf{A}$, giving us the notion of a (simplified) weak (Spector or Kohlenbach) bar recursor within $\mathbf{A}$. We are now able to conclude:

Theorem 31 *No simplified weak Kohlenbach bar recursor within* $\mathbf{A}$ *can be* $\mathrm{T} + \mathit{min}$ *definable.*

PROOF: If BR were a term of $\mathrm{T}+\mathit{min}$ defining a simplified weak Kohlenbach bar recursor in $\mathbf{A}$, then by adequacy of $[\![-]\!]_{\mathbf{A}}$, BR would satisfy the conditions in Theorem 30, a contradiction. □

The corresponding results for Spector bar recursion follow by Proposition 20 relativized to $\mathbf{A}$. It is also clear that we obtain similar results with W in place of $\mathrm{T} + \mathit{min}$.

4.2 Total models

We now consider the situation for total type structures such as Ct and HEO. We work in the general setting of a simply-typed total combinatory algebra $\mathbf{A}$ with $\mathbf{A}(\mathtt{N}) = \mathbb{N}$.

Our formulation for total models will have a character rather different from the above: since no suitable element *min* will be present in $\mathbf{A}$, we cannot induce an interpretation $[\![-]\!]$ straightforwardly from an interpretation of the constants—indeed, there will be terms of $\mathrm{T} + \mathit{min}$ that have no denotation in $\mathbf{A}$. Instead, we resort to an approach more in the spirit of Kleene's original definition of computability in total settings, adapting the treatment in [20]. It is best here to assume that $\mathbf{A}$ is *extensional*: in fact, we shall assume that each $\mathbf{A}(\sigma \to \tau)$ is a set of functions $\mathbf{A}(\sigma) \to \mathbf{A}(\tau)$. It is well-known that this implies that $\mathbf{A}$ is a typed λ-algebra (see [20, Section 4.1]). We shall furthermore assume that $\mathbf{A}$ is a model of T_0: that is, $\mathbf{A}$ contains elements *suc*, *pre*, *ifzero*, and rec_σ for σ of level 0 satisfying the usual defining equations for these constants. (Note that in the total extensional setting, there is no real difference between T_0 and T_0^{str}, or between rec_σ and $\mathit{rec}_\sigma^{str}$, and the operator *byval* is redundant. We shall henceforth use T_0 in this context as it is directly a sublanguage of $\mathrm{T} + \mathit{min}$.)

First, we recall that every $\mathrm{T} + \mathit{min}$ term is $\beta\eta$-equivalent to one in *long* $\beta\eta$*-normal form*—that is, to a β-normal term in which every occurrence of any variable or constant f is fully applied (i.e. appears at the head of a subterm $fN_0 \dots N_{r-1}$ of type $\mathtt{N}$). We shall define a (partial) interpretation in $\mathbf{A}$ for $\mathrm{T} + \mathit{min}$ terms of this kind, and will in general write $\mathrm{nf}(M)$ for the long $\beta\eta$-normal form of M.[9]

For a given term M, a *valuation* ν for M will be a map assigning to each free variable x^σ within M an element $\nu(x) \in \mathbf{A}(\sigma)$. We shall define a partial interpretation assigning to certain terms $M : \sigma$ and valuations ν for M an element $[\![M]\!]_\nu \in \mathbf{A}(\sigma)$. This takes the form of an inductive definition of the relation $[\![M]\!]_\nu = a$, where M is a $\beta\eta$-normal form of some type σ, ν is a valuation for M, and $a \in \mathbf{A}(\sigma)$.

1. $[\![\widehat{n}]\!]_\nu = n$.
2. If $[\![M]\!]_\nu = n$ then $[\![\mathit{suc}\, M]\!]_\nu = n + 1$ and $[\![\mathit{pre}\, M]\!]_\nu = n \dot{-} 1$, where $\dot{-}$ is truncated subtraction.
3. If $[\![M]\!]_\nu = 0$ and $[\![N]\!]_\nu = n$, then $[\![\mathit{ifzero}\, M\, N\, P]\!]_\nu = n$.

[9] The correspondence between β-normal forms and Kleene-style indices is explained in [20, Section 5.1]. Here we use long $\beta\eta$-normal forms in this role because of our treatment of *suc* and rec_σ as first-class constants.

4. If $[\![M]\!]_\nu = m+1$ and $[\![P]\!]_\nu = n$, then $[\![\mathit{ifzero}\, M\, N\, P]\!]_\nu = n$.
5. If $[\![N]\!]_\nu = 0$ and $[\![\mathrm{nf}(X\vec{Y})]\!]_\nu = m$, then $[\![\mathit{rec}_\sigma\, X\, F\, N\, \vec{Y}]\!]_\nu = m$.
6. If $[\![N]\!]_\nu = n+1$ and $[\![\mathrm{nf}(F\,(\mathit{rec}_\sigma\, X\, F\, \widehat{n})\, \widehat{n}\, \vec{Y})]\!]_\nu = m$, then $[\![\mathit{rec}_\sigma\, X\, F\, N\, \vec{Y}]\!]_\nu = m$.
7. If $[\![N]\!]_\nu = n$ and $[\![\mathrm{nf}(F\,\widehat{n})]\!]_\nu = 0$, then $[\![\mathit{min}\, F\, N]\!]_\nu = n$.
8. If $[\![N]\!]_\nu = n$, $[\![\mathrm{nf}(F\,\widehat{n})]\!]_\nu = i+1$ and $[\![\mathit{min}\, F\, \widehat{n+1}]\!]_\nu = m$, then $[\![\mathit{min}\, F\, N]\!]_\nu = m$.
9. If $f \in \mathbf{A}(\sigma \to \tau)$ and $[\![M]\!]_{\nu[y \mapsto a]} = f(a)$ for all $a \in \mathbf{A}(\sigma)$, where $y^\sigma \notin \mathrm{dom}\, \nu$, then $[\![\lambda y.M]\!]_\nu = f$.
10. If $\nu(x) = f$ and $[\![P_i]\!]_\nu = a_i$ for each $i < r$, then $[\![xP_0 \ldots P_{r-1}]\!]_\nu = f(a_0, \cdots, a_{r-1})$.

Other ways of treating the recursors rec_σ would be possible: the definition chosen above errs on the side of generosity, in that it is possible e.g. for $[\![\mathit{rec}_\sigma\, X\, F\, \widehat{0}\, \vec{Y}]\!]_\nu$ to be defined even when $[\![X]\!]_\nu$ is not. Note too that we are not assuming that all the System T operators rec_σ are actually present in $\mathbf{A}$—if they are not, there will of course be many System T terms whose denotations in $\mathbf{A}$ are undefined.

Definition 32 *We say a partial function* $\Phi : \mathbf{A}(\sigma_0) \times \cdots \times \mathbf{A}(\sigma_{r-1}) \rightharpoonup \mathbb{N}$ *is* T + *min* computable *if there is a closed* T + *min term* $M : \mathtt{N}$ *with free variables among* $x_0^{\sigma_0}, \ldots, x_{r-1}^{\sigma_{r-1}}$ *such that for all* $a_0 \in \mathbf{A}(\sigma_0), \ldots, a_{r-1} \in \mathbf{A}(\sigma_{r-1})$ *and all* $n \in \mathbb{N}$ *we have*

$$[\![M]\!]_{x_0 \mapsto a_0, \ldots, x_{r-1} \mapsto a_{r-1}} \;\simeq\; \Phi(a_0, \ldots, a_{r-1})\ ,$$

where $\simeq$ *means Kleene equality.*

Comparing this with the treatment in [20, Section 5.1], it is clear that if we restrict our language to $\mathrm{T}_0 + \mathit{min}$, the computable partial functions over $\mathbf{A}$ obtained as above coincide exactly with Kleene's μ-computable partial functions. We also note in passing that if the language is extended with the operator *Eval* described in [20, Section 5.1], the computable partial functions are exactly the Kleene S1–S9 computable ones.

Next, we may adapt earlier definitions to say what it means to be a bar recursor with respect to $\mathbf{A}$. Note that since $\mathbf{A}$ is a model of T_0^{str}, all functions $[\vec{x}\, j^\omega]$ as defined in Section 2.3 are present in $\mathbf{A}(1)$.

Definition 33 *(i) For any* $F \in \mathbf{A}(2)$*, the* Kohlenbach tree $\mathcal{T}^K(F)$ *consists of all* $\vec{x}$ *such that no proper prefix* $\vec{x}'$ *of* $\vec{x}$ *satisfies* $F([\vec{x}'\, 0^\omega]) = F([\vec{x}'\, 1^\omega])$.

(ii) A partial function $\Phi : \mathbf{A}(2) \times \mathbf{A}(2) \times \mathbf{A}(0) \rightharpoonup \mathbb{N}$ *is a* simplified Kohlenbach bar recursor *if for all* $F, G \in \mathbf{A}(2)$ *with* $\mathcal{T}^K(F)$ *well-founded, and for all* $\vec{x} \in \mathcal{T}^K(F)$*, we have*

$$\begin{array}{lcl} \Phi(F, G, \langle\vec{x}\rangle) & = & 2\langle\vec{x}\rangle + 1 \quad \textit{if } \vec{x} \in \mathcal{T}^K(F)^l\ , \\ \Phi(F, G, \langle\vec{x}\rangle) & = & G(\Lambda z.\, \Phi(F, G, \langle\vec{x}, z\rangle)) \quad \textit{if } \vec{x} \in \mathcal{T}^K(F)^n\ . \end{array}$$

We shall take it to be part of the meaning of the latter condition that the relevant function $\Lambda z.\Phi(F, G, \langle\vec{x}, z\rangle)$ is indeed present in $\mathbf{A}(1)$; this is in effect a further hypothesis on $\mathbf{A}$ which holds in all cases of interest.

We mention a few examples, all of which fall within the scope of Theorem 36 below:

1. In the Kleene-Kreisel model Ct, it is the case for any $F \in \mathsf{Ct}(2)$ that $\mathcal{T}^K(F)$ is well-founded, and indeed a simplified bar recursor Φ is present within Ct itself, as an element of $\mathsf{Ct}(2 \to 2 \to 0 \to 0)$. It is known, furthermore, that such an element is Kleene S1–S9 computable (see [20, Section 8.3]).

2. By contrast, in the model HEO, there are functionals F such that $\mathcal{T}^K(F)$ is not well-founded (such functionals arise from the *Kleene tree* as explained in [20, Section 9.1]), and consequently it is not possible to find a total bar recursor within HEO itself. Nonetheless, partial simplified bar recursors $\Phi : \mathsf{HEO}(2) \times \mathsf{HEO}(2) \times \mathsf{HEO}(0) \rightharpoonup \mathbb{N}$ as defined above do exist and are Kleene computable, by the same algorithm as for Ct. The situation is in fact precisely similar for the full set-theoretic model S: thus, partial bar recursors in the spirit of Definition 15 are Kleene computable over S.

3. Another model of a quite different character is Bezem's type structure of *strongly majorizable functionals* [5], which has been found to be valuable in proof theory. Here, as in Ct, a bar recursor lives as a total object within the model itself—despite the presence of *discontinuous* type 2 elements in the model.

We shall show that no simplified Kohlenbach bar recursor for $\mathbf{A}$ can be $\mathrm{T} + \mathit{min}$ computable in the sense above. This will follow easily from Theorem 30 once we have established the 'adequacy' of our partial interpretation $[\![-]\!]$. This we do by means of a standard logical relations argument. For each σ, let us define a relation $R_\sigma(M, a)$ between closed $\mathrm{T} + \mathit{min}$ terms $M : \sigma$ and elements $a \in \mathbf{A}(\sigma)$ as follows:

- $R_{\mathbb{N}}(M, m)$ iff $M \rightsquigarrow^* \widehat{m}$.
- $R_{\sigma\to\tau}(M, f)$ iff for all $N : \sigma$ and $a \in \mathbf{A}(\sigma)$, $R_\sigma(N, a)$ implies $R_\tau(MN, f(a))$.

We often omit the type annotations and may refer to any of the R_σ as R.

Lemma 34 *If $[\![M]\!]_\nu = a$ and $R(N_i, \nu(x_i))$ for all x_i free in M, then $R(M[\vec{x} \mapsto \vec{N}], a)$.*

Proof: By induction on the generation of $[\![M]\!]_\nu = a$ via clauses 1–10 above. The cases for clauses 1–4 are trivial, and those for clauses 5–8 are very straightforward, using the fact that any term M is observationally equivalent to $\mathrm{nf}(M)$ by the context lemma.

For clause 9, suppose we have $[\![\lambda y^\sigma.M]\!]_\nu = f \in \mathbf{A}(\sigma \to \tau)$ arising from $[\![M]\!]_{\nu[y\mapsto a]} = f(a)$ for all $a \in \mathbf{A}(\sigma)$, and suppose also that $R(N_i, \nu(x_i))$ for all x_i free in $\lambda x.M$. We wish to show that $R_{\sigma\to\tau}((\lambda y.M)[\vec{x} \mapsto \vec{N}], f)$: that is, that for all $P : \sigma$ and $a \in \mathbf{A}(\sigma)$, $R_\sigma(P, a)$ implies $R_\tau((\lambda y.M)[\vec{x} \mapsto \vec{N}](P), f(a))$. So suppose $R_\sigma(P, a)$. By assumption, we have $[\![M]\!]_{\nu[x\mapsto a]} = f(a)$ and $R(N_i\, \nu(x_i))$ for all i, so $R_\tau(M[\vec{x} \mapsto \vec{N}, y \mapsto P],\ f(a))$ by the induction hypothesis. The desired conclusion follows, since

$$(\lambda y.M)[\vec{x} \mapsto \vec{N}](P) \ \rightsquigarrow\ M[\vec{x} \mapsto \vec{N}, y \mapsto P]$$

and it is easy to see by induction on types that if $Q \rightsquigarrow Q'$ and $R(Q', b)$ then $R(Q, b)$.

For clause 10, suppose we have $[\![x_j \vec{P}]\!]_\nu = f(\vec{a})$ arising from $\nu(x_j) = f$ and $[\![P_i]\!]_\nu = a_i$ for each i, and suppose again that $R(N_i, \nu(x_i))$ for all i. Writing * for the substitution

$[\vec{x} \mapsto \vec{N}]$, we have $(x_j \vec{P})^* = N_j \vec{P}^*$, so it will suffice to show that $R(N_j \vec{P}^*, f(\vec{a}))$. But we have $R(N_j, \nu(x_j))$ where $\nu(x_j) = f$, and also $R(P_i^*, a_i)$ for each i by the induction hypothesis, so by definition of R_σ where σ is the type of N_j, we have $R(N_j \vec{P}^*, f(\vec{a}))$ as required. $\square$

The converse to Lemma 34 is not true. For instance, if $M = f(min\,(\lambda y.\widehat{1})\,\widehat{0})$, $N = \lambda x.\widehat{2}$ and $a = \Lambda x.2 \in \mathbf{A}(1)$, then $R_1(N, a)$ and $M[f \mapsto N] \rightsquigarrow^* \widehat{2}$, but $[\![M]\!]_{f \mapsto a}$ is undefined because $min\,(\lambda y.\widehat{1})\,\widehat{0}$ receives no denotation. In this sense, $\mathrm{T} + min$ computability in a total model is a stricter condition than it would be in a partial model. It is therefore not too surprising that no bar recursor for a total model can be $\mathrm{T} + min$ computable.

Recall that we are assuming that $\mathbf{A}$ is a model of T_0, in the sense that $\mathbf{A}$ contains suitable elements *suc*, *pre*, *ifzero*, rec_σ satisfying the relevant equations, giving rise via the λ-algebra structure of $\mathbf{A}$ to an interpretation of T_0 which we shall denote by I. We may now verify that our two ways of interpreting T_0 terms are in accord:

Lemma 35 *Suppose M is any long $\beta\eta$-normal T_0 term, and $\nu = (\vec{x} \mapsto \vec{a})$ is any valuation for M. Then $[\![M]\!]_\nu$ is defined and is equal to $I_{\vec{x}}(M)(\vec{a})$.*

Proof: A routine induction on the structure of M. $\square$

We now have all the pieces needed for the main result, which establishes Corollary 6.3.33 of [20]. It is worth noting that the proof of the following theorem makes use of a *bar induction* principle at the meta-level, namely that if $\mathcal{T}$ is a well-founded tree as defined in Subsection 2.3, then

$$(\forall \vec{x} \in \mathcal{T}^l.\, P(\vec{x})) \;\wedge\; (\forall \vec{x} \in \mathcal{T}^n.\, (\forall y \in \mathbb{N}.\, P(\vec{x}, y)) \;\Rightarrow\; P(\vec{x})) \;\Rightarrow\; (\forall \vec{x} \in \mathcal{T}.\, P(\vec{x}))$$

(for a certain predicate P to be specified in the course of the proof). This is of course unproblematic if our metatheory is classical; there may also be weaker metatheories that suffice for our arguments, but we will not enter into an investigation of this here.

Theorem 36 *No simplified Kohlenbach bar recursor for $\mathbf{A}$ can be $\mathrm{T} + min$ computable.*

Proof: Suppose B were a $\mathrm{T} + min$ term with free variables $F : 2$, $G : 2$, $x : 0$ defining a simplified Kohlenbach bar recursor $\Phi : \mathbf{A}(2) \times \mathbf{A}(2) \times \mathbf{A}(0) \rightharpoonup \mathbb{N}$ as above. We claim that $\mathsf{BR} = \lambda FGx.B$ satisfies the conditions of Theorem 30, yielding a contradiction. Indeed, suppose $\hat{F}, \hat{G} : 2$ are closed T_0^{str} terms with $\mathcal{T}^K(\hat{F})$ well-founded. Construing $\hat{F}, \hat{G}$ as T_0 terms, we obtain elements $[\![\hat{F}]\!], [\![\hat{G}]\!] \in \mathbf{A}(2)$ by Lemma 35, and it is clear from Lemma 34 that $\mathcal{T}^K([\![\hat{F}]\!]) = \mathcal{T}^K(\hat{F})$.

Now suppose $\vec{x} \in \mathcal{T}^K(\hat{F})$. We now show by meta-level bar induction on $\vec{x} \in \mathcal{T}^K(\hat{F})$ that for all such $\vec{x}$, $[\![B]\!]_{F \mapsto [\![\hat{F}]\!],\, G \mapsto [\![\hat{G}]\!],\, x \mapsto \langle \vec{x} \rangle}$ is defined and agrees with the value of $\mathsf{BR}\,\hat{F}\,\hat{G}\,\widehat{\langle \vec{x} \rangle}$, and moreover the latter satisfies the relevant condition of Theorem 30.

First, if $\vec{x} \in \mathcal{T}^K(\hat{F})^l$, then by Definitions 32 and 33 we have

$$[\![B]\!]_{F \mapsto [\![\hat{F}]\!],\, G \mapsto [\![\hat{G}]\!],\, x \mapsto \langle \vec{x} \rangle} \;=\; \Phi\,([\![\hat{F}]\!], [\![\hat{G}]\!], \langle \vec{x} \rangle) \;=\; 2\langle \vec{x} \rangle + 1\ .$$

Now by Lemma 34 we have $R(\hat{F}, [\![\hat{F}]\!])$, $R(\hat{G}, [\![\hat{G}]\!])$ and $R(\widehat{\langle \vec{x} \rangle}, \langle \vec{x} \rangle)$, so by the same again we have $R_{\mathtt{N}}\,(B[F \mapsto \hat{F}, G \mapsto \hat{G}, x \mapsto \widehat{\langle \vec{x} \rangle}],\, 2\langle \vec{x} \rangle + 1)$, meaning that

$$B[F \mapsto \hat{F}, G \mapsto \hat{G}, x \mapsto \widehat{\langle \vec{x} \rangle}] \;\rightsquigarrow^*\; \widehat{2\langle \vec{x} \rangle + 1}\,.$$

Hence $\mathsf{BR}\,\hat{F}\,\hat{G}\,\widehat{\langle \vec{x} \rangle}$ satisfies the first condition of Theorem 30, and all parts of the induction claim are established.

Now suppose that $\vec{x} \in \mathcal{T}^K(\hat{F})^n$, where each child $\vec{x}, z$ satisfies the induction claim. We first show that

$$R_{\mathtt{N}\to\mathtt{N}}\;(\lambda z.\,\mathsf{BR}\;\hat{F}\;\hat{G}\;(\mathrm{add}\;\widehat{\langle \vec{x} \rangle}\;z),\;\; \Lambda z.\,[\![B]\!]_{F\mapsto[\![\hat{F}]\!],\,G\mapsto[\![\hat{G}]\!],\,x\mapsto\langle \vec{x},z\rangle})\,.$$

For this, it suffices to show that if $z \in N$ and $R_{\mathtt{N}}(Z, z)$ (i.e. $Z \rightsquigarrow^* \widehat{z}$), then

$$R_{\mathtt{N}}\;((\lambda z.\,\mathsf{BR}\;\hat{F}\;\hat{g}\;(\mathrm{add}\;\widehat{\langle \vec{x} \rangle}\;z))Z,\;\; [\![B]\!]_{F\mapsto[\![\hat{F}]\!],\,G\mapsto[\![\hat{G}]\!],\,x\mapsto\langle \vec{x},z\rangle})\,.$$

But this holds by the induction hypothesis along with the observational equivalence

$$(\lambda z.\,\mathsf{BR}\;\hat{F}\;\hat{G}\;(\mathrm{add}\;\widehat{\langle \vec{x} \rangle}\;z))Z \;\simeq_{\mathrm{obs}}\; \mathsf{BR}\;\hat{F}\;\hat{G}\;\widehat{\langle \vec{x}, z \rangle}\,.$$

Since $R(\hat{G}, [\![\hat{G}]\!])$, we may conclude that

$$R_{\mathtt{N}}\;(\hat{G}\;(\lambda z.\,\mathsf{BR}\;\hat{F}\;\hat{G}\;(\mathrm{add}\;\widehat{\langle \vec{x} \rangle}\;z)),\;\; [\![\hat{G}]\!]\;(\Lambda z.\,[\![B]\!]_{F\mapsto[\![\hat{F}]\!],\,G\mapsto[\![\hat{G}]\!],\,x\mapsto\langle \vec{x},z\rangle}))$$

so that the term on the left evaluates to (the numeral for) the value on the right. But also by Definitions 32 and 33 we have

$$\begin{aligned}
[\![B]\!]_{F\mapsto[\![\hat{F}]\!],\,G\mapsto[\![\hat{G}]\!],\,x\mapsto\langle \vec{x}\rangle} &= \Phi\,([\![\hat{F}]\!], [\![\hat{G}]\!], \langle \vec{x} \rangle) \\
&= [\![\hat{G}]\!]\;(\Lambda z.\,\Phi\,([\![\hat{F}]\!], [\![\hat{G}]\!], \langle \vec{x}, z \rangle)) \\
&= [\![\hat{G}]\!]\;(\Lambda z.\,[\![B]\!]_{F\mapsto[\![\hat{F}]\!],\,G\mapsto[\![\hat{G}]\!],\,x\mapsto\langle \vec{x},z\rangle})\,.
\end{aligned}$$

In particular, $[\![B]\!]_{F\mapsto[\![\hat{F}]\!],\,G\mapsto[\![\hat{G}]\!],\,x\mapsto\langle \vec{x}\rangle}$ is defined, so using Lemma 34 as before, we see that $B[F \mapsto \hat{F}, G \mapsto \hat{G}, x \mapsto \widehat{\langle \vec{x} \rangle}]$ also evaluates to $[\![\hat{G}]\!]\;(\Lambda z.[\![B]\!]_{F\mapsto[\![\hat{F}]\!],\,G\mapsto[\![\hat{G}]\!],\,x\mapsto\langle \vec{x},z\rangle})$. Thus the induction claim is established for $\vec{x}$.

We have thus shown that BR satisfies the conditions of Theorem 30, so a contradiction with that theorem is established. □

Clearly, similar results hold for Spector bar recursion or for the language W.

References

[1] Abramsky, S., Jagadeesan, R. and Malacaria, P.: Full abstraction for PCF. Information and Computation **163(2)**, 409–470 (2000)

[2] Berardi, S., Oliva, P., Steila, S.: An analysis of the Podelski-Rybalchenko termination theorem via bar recursion. Journal of Logic and Computation, published online (2015)

[3] Berger, U.: Minimization vs. recursion on the partial continuous functionals. In: Gärdenfors, P., Woleński, J., Kijania-Placek, K. (eds.), In the Scope of Logic, Methodology and Philosophy of Science, Cracow, August 1999, pp. 57-64. Kluwer, Dordrecht (2002)

[4] Bergstra, J.: Continuity and Computability in Finite Types. PhD thesis, University of Utrecht (1976)

[5] Bezem, M.: Strongly majorizable functionals of finite type: a model for bar recursion containing discontinuous functionals. Journal of Symbolic Logic **50**, 652–660 (1985).

[6] Brouwer, L.E.J.: Über Definitionsbereiche von Funktionen (On the domains of definition of functions). Mathematische Annalen **97**, 60–75 (1927). English translation in L.E.J. Brouwer, Collected Works, Volume I, ed. A. Heyting *et al*, North-Holland (1975), pp. 390–405.

[7] Escardó, M.H. and Oliva, P.: Computing Nash equilibria of unbounded games. In: Proceedings of the Turing Centenary Conference, EPiC Series **10**, 53-65, Manchester (2012)

[8] Escardó, M.H. and Oliva, P.: Bar recursion and products of selection functions. Journal of Symbolic Logic **80(1)**, 1–28 (2015)

[9] Hedges, J., Oliva, P., Winschel, E., Winschel, V. and Zahn, P.: Higher-order decision theory. In: Proceedings of the 5th International Conference on Algorithmic Decision Theory (2017)

[10] Hyland, J.M.E., Recursion Theory on the Countable Functionals. DPhil thesis, University of Oxford (1975)

[11] Hyland, J.M.E. and Ong, C.-H.L.: On full abstraction for PCF: I, II and III. Information and Computation **163**, 285–408 (2000)

[12] Kleene, S.C.: Recursive functionals and quantifiers of finite types I, Transactions of the American Mathematical Society **91(1)**, 1–52 (1959)

[13] Kohlenbach, U.: Theory of Majorizable and Continuous Functionals and their Use for the Extraction of Bounds from Non-Constructive Proofs: Effective Moduli of Uniqueness for Best Approximations from Ineffective Proofs of Uniqueness. PhD thesis, Frankfurt (1990)

[14] Kohlenbach, U.: Applied Proof Theory: Proof Interpretations and their Use in Mathematics. Springer (2008)

[15] Kreisel, G.: Interpretation of analysis by means of constructive functionals of finite types. In: Heyting, A. (ed.), Constructivity in Mathematics: Proceedings of the Colloquium held in Amsterdam, 1957, pp. 101-128. North-Holland, Amsterdam (1959)

[16] Kreisel, G.: Set theoretic problems suggested by the notion of potential totality. In: Infinitistic Methods, Proceedings of the Symposium on Foundations of Mathematics (Warsaw 1959), pp. 103-140. Pergamon Press, Oxford (1961)

[17] Longley, J.R.: Bar recursion is not T+min definable. Informatics Research Report EDI-INF-RR-1420, University of Edinburgh (2015)

[18] Longley, J.R.: The recursion hierarchy for PCF is strict. To appear in Logical Methods in Computer Science; available at `arxiv.org/abs/1607.04611` (2018). Earlier version as Informatics Research Report EDI-INF-RR-1421, University of Edinburgh (2015)

[19] Longley, J.R.: On the relative expressive power of some sublanguages of PCF. In preparation.

[20] Longley, J. and Normann, D.: Higher-Order Computability. Theory and Applications of Computability, Springer (2015)

[21] Milner, R.: Fully abstract models of typed λ-calculi. Theoretical Computer Science **4(1)**, 1–22 (1977)

[22] Oliva, P. and Powell, T.: A constructive interpretation of Ramsey's theorem via the product of selection functions. Mathematical Structures in Computer Science **25(8)**, 1755–1778 (2015)

[23] Plotkin, G.D.: LCF considered as a programming language. Theoretical Computer Science **5(3)**, 223–255 (1977)

[24] Plotkin, G.D.: Full abstraction, totality and PCF. Mathematical Structures in Computer Science **9(1)**, 1–20 (1999)

[25] Sazonov, V.Yu.: Expressibility in D. Scott's LCF language. Algebra and Logic **15(3)**, 192–206 (1976)

[26] Scarpellini, B.: A model for barrecursion of higher types. Compositio Mathematica **23**, 123–153 (1971)

[27] Spector, C.: Provably recursive functionals of analysis: a consistency proof of analysis by an extension of principles formulated in current intuitionistic mathematics. In: Dekker, J. (ed.) Recursive Function Theory, Proceedings of Symposia in Pure Mathematics, Volume 5, pp. 1–27. AMS, Providence (1962)

[28] Troelstra, A.S. (ed.): Metamathematical Investigation of Intuitionistic Arithmetic and Analysis. Springer, Berlin (1973)